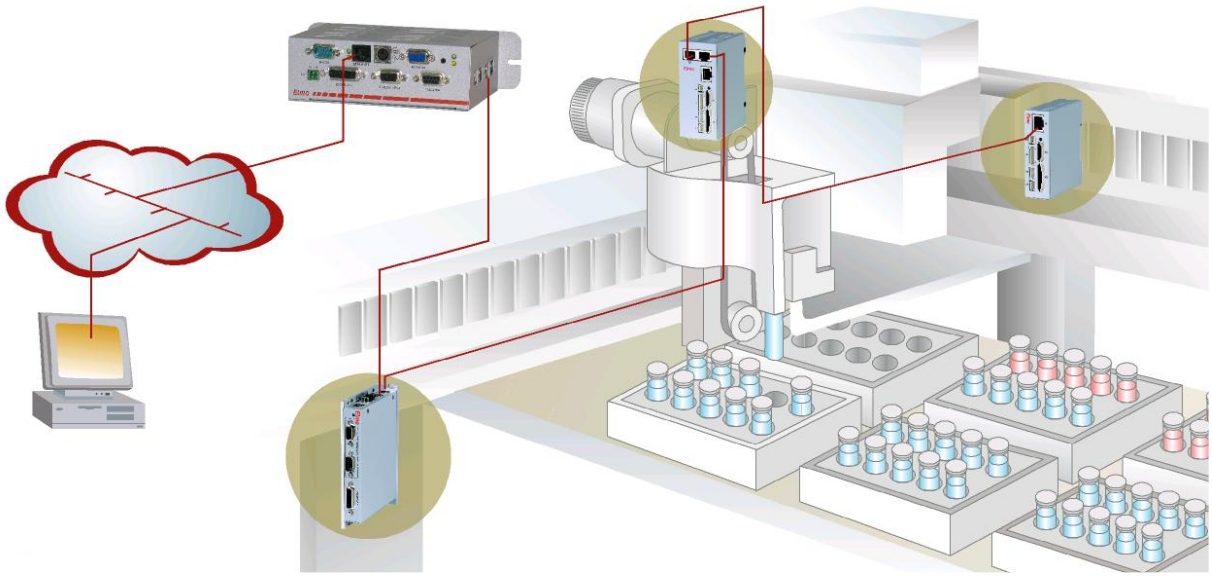
Maestro

Software Manual



Draft

February 2005

Elmo
Motion Control

Important Notice

This guide is delivered subject to the following conditions and restrictions:

- This guide contains proprietary information belonging to Elmo Motion Control Ltd. Such information is supplied solely for the purpose of assisting users of *Maestro* motion supervisor.
- The text and graphics included in this manual are for the purpose of illustration and reference only. The specifications on which they are based are subject to change without notice.
- Information in this document is subject to change without notice.

Doc. No. MAN-MASSW
Copyright © 2005
Elmo Motion Control
All rights reserved

Revision History

Ver. A	February 2005	First Draft	MAN-MASSW.pdf)

Elmo Motion Control Inc.
1 Park Drive, Suite 12
Westford, MA 01886 USA
Tel: +1 (978) 399-0034
Fax: +1 (978) 399-0035
email: info-us@elmomc.com

Elmo Motion Control GmbH
Steinbeisstrasse 41
D-78056, Villingen-Schwenningen Germany
Tel: +49 (07720) 8577-60
Fax: +49 (07720) 8577-70
email: info-de@elmomc.com

www.elmomc.com

Contents

Chapter 1: Introduction 1-1
1.1 Maestro Highlights 1-1
1.2 Supplementary Documents 1-2
1.3 Command Specification 1-3
1.4 Scope 1-3

Chapter 2: Functional Overview 2-1
2.1 Functional Block Diagram 2-1
2.2 Host Communications Services 2-2
2.3 Command Line Interpreter 2-2
2.4 The Kernel 2-2
2.5 Motion Manager 2-3
2.6 CANopen Network Communications Services 2-3

Chapter 3: Program Editor Menus 3-1
3.1 The Program Editor Desktop 3-2
3.1.1 Desktop Windows 3-2
3.1.2 Program Editor Toolbars 3-3
3.1.3 The Menu Bar 3-5
3.1.4 Workspace Menus (accessed with right mouse button) 3-8
3.1.5 Resource Editing Menus (right mouse button only) 3-9
3.1.6 Customizing the Elmo Studio 3-10
3.1.6.1 The Customize Dialog Boxes 3-10
3.1.6.2 The Options Dialog Boxes 3-11
3.1.6.3 Convert to New Format 3-12

Chapter 4: Using the Program Editor 4-1
4.1 Creating a Workspace 4-1
4.2 Setting up the Resources 4-2
4.3 Opening an Existing Program 4-3
4.4 Editing a Program File 4-4
4.5 Compiling a Program 4-4
4.6 Building a Program 4-4
4.7 Running a Program 4-4
4.8 Debugging 4-5

Chapter 5: Host Communications 5-1
5.1 Setting Up the Host through Elmo's Studio 5-1
5.2 Verifying or Changing the Host 5-1
5.3 Choosing the Host through the Composer 5-2

Chapter 6: Command Line Interface 6-1
6.1 Accessing the CLI 6-1

Chapter 7: Motion Instructions 7-1
7.1 Axis 7-1
7.1.1 Axis Motion Commands 7-1
7.1.2 Axis Properties 7-2
7.1.3 Axis Functions 7-3

7.2 Group 7-5
7.2.1 Group Motion Commands 7-5
7.2.2 Group Properties 7-5
7.2.3 Group Functions 7-6
7.3 Vector 7-7
7.3.1 Vector Motion Commands 7-7
7.3.2 Vector Properties 7-8
7.3.3 Vector Functions 7-12
7.4 General Instructions 7-17

Chapter 8: MAXL Program Language 8-1
8.1 Lexical Conventions 8-1
8.1.1 Comments 8-1
8.1.2 Identifiers 8-2
8.1.3 MAXL Keywords 8-3
8.1.4 Punctuators 8-3
8.1.5 Operators 8-3
8.1.6 Literals 8-5
8.1.6.1 Integer Constant 8-5
8.1.6.2 Floating-Point Constant 8-6
8.1.6.3 String Literals 8-8
8.2 Basic Concepts 8-8
8.2.1 Declarations and Definitions 8-8
8.2.1.1 Declarations 8-8
8.2.1.2 Definitions 8-9
8.2.2 Program 8-9
8.2.3 Startup and Termination 8-9
8.2.3.1 Program Startup – the run Function 8-9
8.2.3.2 Program Termination 8-10
8.2.4 Types 8-10
8.2.4.1 Fundamental Types 8-11
8.2.4.2 Object types 8-11
8.3 Standard Conversions 8-11
8.4 Expressions 8-12
8.4.1 Types of Expressions 8-12
8.4.1.1 MAXL Primary Expressions 8-13
8.4.1.2 Postfix Expressions 8-14
8.4.1.3 Expressions with Unary Operators 8-15
8.4.1.4 Expressions with Binary Operators 8-16
8.4.1.5 MAXL Logical Operators 8-22
8.4.1.6 Simple Assignment 8-23
8.4.2 Semantics of Expressions 8-23
8.4.2.1 Order of Evaluation 8-23
8.5 Statements 8-24
8.5.1 Labeled Statements 8-24
8.5.1.1 Using Labels with the goto Statement 8-25
8.5.1.2 Using Labels in the case Statement 8-25
8.5.2 Selection Statements 8-26
8.5.2.1 The MAXL if Statement 8-26
8.5.2.2 The MAXL switch Statement 8-26

8.5.3 Iteration Statements 8-28
8.5.3.1 The MAXL while Statement 8-28
8.5.3.2 The MAXL for Statement 8-29
8.5.4 Jump Statements 8-29
8.5.4.1 The MAXL break Statement 8-30
8.5.4.2 The MAXL continue Statement 8-30
8.5.4.3 The MAXL return Statement 8-30
8.5.4.4 The goto Statement 8-31
8.5.5 Declaration Statements 8-31
8.6 Functions 8-31
8.6.1 Function definition 8-31
8.6.2 Build-in functions 8-33
8.6.3 Callback (interrupt) functions 8-34
8.7 Virtual Machine Control Statements 8-35
8.7.1 wait control statement 8-35
8.7.2 TRACE control statement 8-36
8.7.3 reset control statement 8-36

Chapter 9: The Maestro API 9-1
MAC_Initialize 9-1
MAC_Uninitialize 9-2
MAC_CreateTCPConnection 9-3
MAC_CreateRS232Connection 9-4
MAC_CloseConnection 9-6
MAC_SendCommand 9-7
MAC_LocateDevices 9-8
MAC_GetDevice 9-9
MAC_GetIpByName 9-10
MAC_IsDevicePresent 9-11
MAC_LocateObjects 9-12
MAC_GetObject 9-13
MAC_DownloadTrajectory 9-14
MAC_RemoveTrajectory 9-15
MAC_DownloadProgram 9-16
MAC_RemoveProgram 9-17
MAC_DownloadResources 9-18
MAC_UploadLog 9-19
MAC_GetLastError 9-20
MAC_InitEvents 9-21
MAC_DeinitEvents 9-21
MAC_RegCloseCallback 9-22

Chapter 10: The Recorder 10-1
10.1 Accessing the Recorder 10-1

Appendix A: Setting up the Demo Case A-1
A.1 Setting Up the CAN nodes A-1
A.2 Checking the CANopen Network A-2

Appendix B: Sample Programs B-1
B.1 Elmo Logo Outline Sample B-1
B.2 Line Sample B-1
B.3 Circle Sample B-1
B.4 Circle-to-Line Sample B-2
B.5 Line-to-Circle Sample B-2
B.6 Line-to-Line Sample B-3
B.7 Polygon Sample B-3
B.8 If Sample B-4
B.9 If-Else Sample B-4
B.10 If-Else-If Sample B-4
B.11 For Sample B-4
B.12 While Sample B-5
B.13 Switch Sample B-5
B.14 Break-Continue-Return Sample B-5
B.15 Call Sample B-6
B.16 Homing Sample B-6
B.17 Program Call Sample B-6
B.18 Label and GoTo Sample B-7
B.19 Order of Processing Sample B-7
B.20 Global Variable Sample B-8

Appendix C: Performance Considerations C-1

Chapter 1: Introduction

Elmo's Maestro is a network-based multi-axis motion supervisor that operates in conjunction with Elmo intelligent servo drives to provide a full multi-axis motion control solution. The Maestro and the SimplIQ servo drives share the motion processing workload in a distributed motion control architecture.

1.1 *Maestro* Highlights

The Maestro operates as a Multi-Axis Motion Supervisor to:

- coordinate motion between various axes in synchronized interpolated mode
- integrate event handling into motion control procedures

The Maestro operates as a CANopen Network Node Master for:

- Network management (NMT)
- Clock synchronization
- Network Configuration

The Maestro operates as a Ethernet - CAN gateway

The Maestro acts as a file archiver and distributor of:

- Firmware – Maestro and intelligent drives
- Multi-Axis User Applications – Maestro and intelligent drives
- System Resources

The Maestro operates as a Multi-Axis Motion Analysis & Development tool:

- Multi-Axis recording and analysis tools
- Multi-Axis application development environment

The Maestro can be monitored through a web browser.

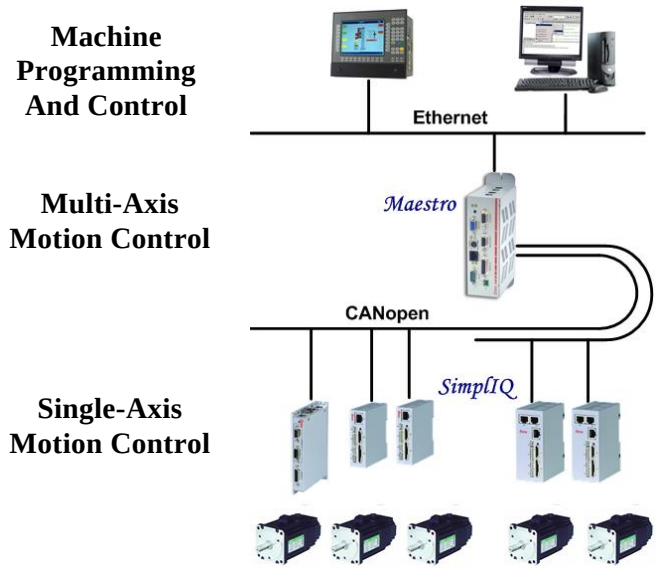


Figure 1-1 Maestro Multi-Axis Control Architecture

1.2 Supplementary Documents

This manual is part of a documentation set that, together, can be used to set up and program the motion of any machine whose motors are controlled by Elmo *SimplIQ* servo drives. Before you can use this manual you will need to carefully follow the instructions in the *Maestro Installation Guide* to set up your Maestro.

The software described in this manual is provided on the CD that accompanies the Maestro or as a download from Elmo's web site. In this manual it is assumed that you have followed the software setup instructions in the *Maestro Installation Guide* and have successfully installed the software.

At least one drive needs to be connected to the Maestro in order for it to function as a motion controller. The *SimplIQ* manuals shown below explain how to set up and program servo drives. Please read the *Installation Guide* that arrived with your servo drive before setting it up. Servo drives are power devices so be careful.

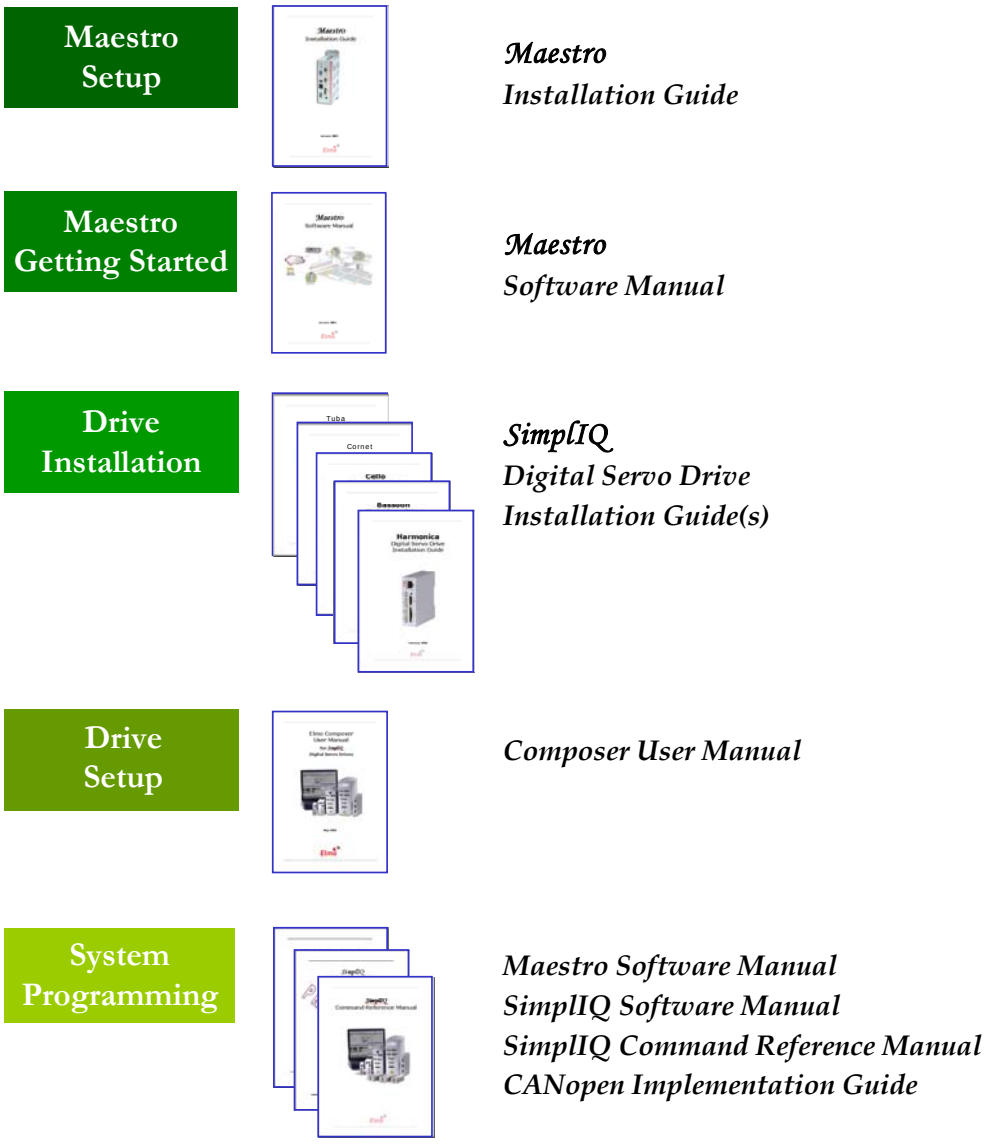


Figure 1-2: **Elmo Documentation Hierarchy**

1.3 Command Specification

Commands for *SimplIQ* drives may be specified from the following sources:

- User program — A program loaded to the servo drive via one of the communication options. After program execution begins, the program is managed by the drive.
- RS-232 — Serial, point-to-point, short-range communication. Although this method is rather slow, RS-232 is very easy to use and requirements are minimal: a standard PC with serial port and ASCII terminal software.
- CANopen — Serial, multi-drop, medium speed and medium-range communication. This type of communication requires special-purpose host hardware and software.

This manual describes the *Maestro* commands that can be specified from each of these sources. Most of the commands are equally available for all three sources. Certain commands, however, are limited in scope according to type of program or communication.

All the commands are available to CAN communication in text form through the OS service, objects 0x1023 and 0x1024. In addition, the numerical set/get commands are available to CAN users in short PDO form, called the "binary interpreter." The binary and the OS SCAN interpreters are described fully in the CAN manual.

CANopen may also be used to manipulate the drive using the object dictionary (OD) method, which is the native CAN method. This manual does not cover OD manipulations with CANopen; refer to the "Object Dictionary" section of the CANopen manual for full explanations.

The *Maestro* drive responds to many privileged commands — such as those used by the Composer setup wizard — that are not documented in this manual.

1.4 Scope

This manual includes the complete list of commands used by the *Maestro* servo drives.

The commands are presented in two ways:

- A task-related listing
- Alphabetically

In the task-related reference, the commands are sorted into groups of related commands. Each group is presented in a table listing the commands with basic descriptions. The alphabetical command listing provides a detailed explanation of each command, with examples and references to the *SimplIQ Software Manual* when necessary.

Chapter 2: Functional Overview

This chapter takes a look at the organization of *Maestro* software.

2.1 Functional Block Diagram

The Maestro's functionality can be organized into the 5 groups shown below.

The first group (*Host Communications Services*) contains the standard interfaces and protocols that enable the *Maestro* to communicates with the "*outside world*".

The *Command Line Interpreter* is a utility that enables individual commands to be executed immediately by either the *Maestro* or by a *SimplIQ* drive on a specified axis.

The *Kernel* is the part of the Maestro that executes user programs.

The *Motion Manager* sends commands and information to all axes and receives information so that it can coordinate motion between all the axes.

The Maestro is designed to manage multiple axes on a CANopen network. The CAN Network Communication Server contains the CANopen interfaces and protocols that enable the *Maestro* to do so.

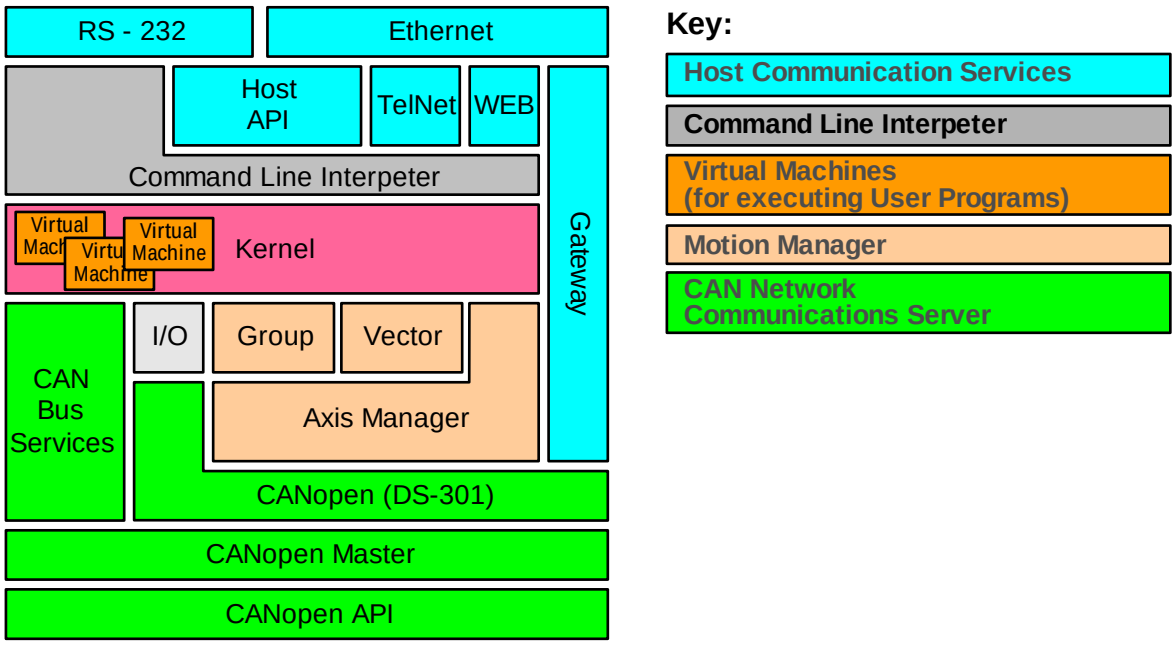


Figure 2-1 The *Maestro's* Building Block

2.2 Host Communications Services

A host application can access the Maestro using either a TCP/IP or RS-232 services. Processes carried out through host communication include:

- Transfer of operating instructions (e.g. for running a program or killing it) to the Maestro
- Transfer operational data (such as the trajectory of the next motion)
- Status requests
- Debugging
- Generation of a "transparent path" from a Composer program to any single end-unit
- Host communication is used to execute different tasks, including:
  - Processing of interpreter commands
  - Maintenance and file download/upload
  - Processing of direct-axis interpreter commands
  - CANopen gateway

2.3 Command Line Interpreter

CLI commands that are sent to the Maestro are either executed by the Maestro itself or are forwarded directly to the specified axis for immediate execution.

The CLI currently supports the following commands:

- Initialization commands
- Commands for collect information
- Axis commands
- Vector commands
- Group command

2.4 The Kernel

- Maestro User Programs can be executes by the Kernel's Virtual Machines.
- The Virtual Machine enables multi-axis programming.
- Each task (program) can work independently of the other tasks
- Communicate and synchronize among themselves can be done by using global variables.
- Multiple task can be used to run different machine functions in parallel.

2.5 Motion Manager

- Axis Manager
- Group
- Vector

2.6 CANopen Network Communications Services

- CANopen Bus services
- CANopen DS 301 Protocol
- CANopen Master
- CANopen API

Chapter 3: Program Editor Menus

Elmo's Program Editor is used for creating and editing multi-axis applications on the *Maestro*. It includes a range of program creation, editing, build and debugging functions. You can use it in conjunction with the Composer to edit your application programs for subsequent download to the servo drive. Elmo's Program Editor is similar in look and general functionality to Microsoft's Visual Studio. As with MS Studio, users should already be familiar with the basics of source file coding, compiling and debugging.

To access the Elmo's Program Editor, click the button or select **Start – Programs - Program Editor.** The Elmo Studio desktop will be displayed as in the following example (once you've opened a program and a workspace):

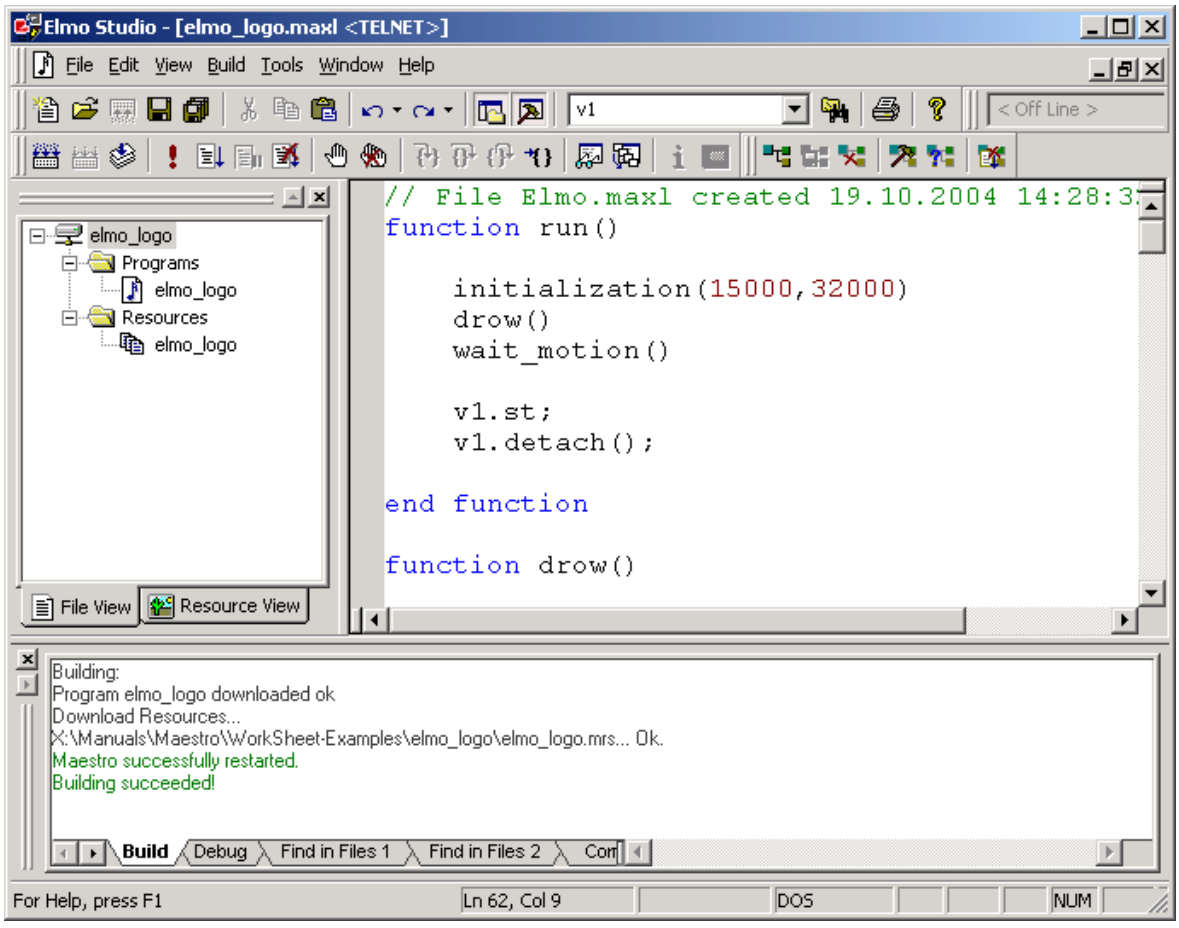


Figure 3-1 The Maestro's Program Editor

This chapter briefly describes the menus, toolbars and functionality of Elmo's Program Editor. You may wish to consult the relevant Elmo *Software Manual, Composer User Manual or SimplIQ Command Reference* for more information about specific program structures, definitions and limits of Elmo products.

3.1 The Program Editor Desktop

Upon accessing Elmo's Program Editor, you will see a set of open windows that can be opened and closed, and manipulated as needed. Across the top, as in most Windows applications, is the menu bar, with movable and customizable toolbars beneath it.

3.1.1 Desktop Windows

The following windows provide on-going information as you work in the Program Editor:

- Workspace window
  This is where the program being edited, compiled or debugged is displayed. In order to display multiple parts of this file at the same time, select **Window - New Window**. An additional pane will be displayed for you to view a different part of the same program.
  To open and close this window toggle the button.

- Stack window
  During debugging, all called functions that have not yet been returned are displayed here.
  To open and close this window toggle the button.

- Output window
  The Program Editor displays processing messages as follows:
  - Build tab
    Status messages from compiler and other tools during a build
  - Debug tab
    Messages from debugger to indicate run-time and other errors
  - Find in tabs
    Search results are displayed in the Find in 1 tab; subsequent searches can be displayed in the Find in 2 tab.

  To open and close this window toggle the button.

- Watch window
  This window provides a view of specified variables (that can be dragged-and-dropped from the Program window), along with their current values as they exist at the time the program is suspended.
  To open and close this window toggle the button.

3.1.2 Program Editor Toolbars

The four toolbars — Standard, Communication, Build and Workspace—contain buttons that enable you to quickly access the most frequently-used tools and options in the Program Editor. You can move the toolbars around the desktop and relocate them for your convenience. You can also remove buttons and add others for commands that you frequently use (see section 3.1.6.1).

Figure 3-2: ***Elmo Program Editor Toolbars***

Tables 4-1 and 4-2 list each toolbar element and its function.

Button/List	Function
	Create a new program.
	Open an existing program.
	Upload a program from connected drive.
	Save the currently open program.
	Cut text from the program.
	Copy selected text in the program.
	Paste text into the program.
	Undo last action.
	Redo last “undo.”
	Display/Hide (Toggle) the workspace
	Display/Hide (toggle) the Output window.
	Find displayed item.
	Find all occurrences of selected item.
	Print program.
	Get context-sensitive help.

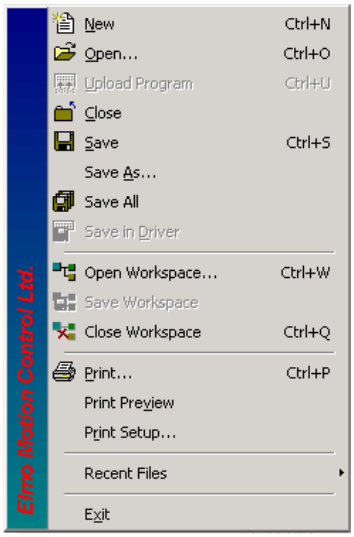
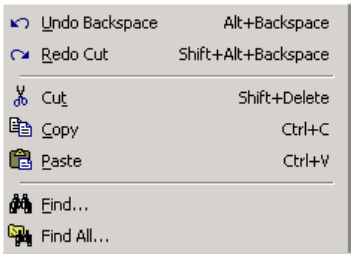
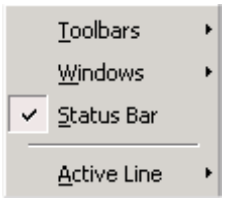
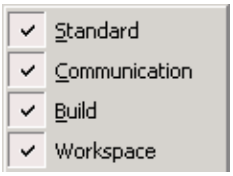
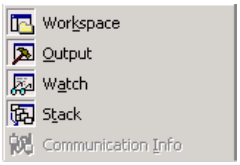
Table 3-1: ***Standard Toolbar Elements***

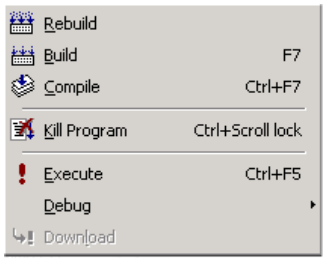
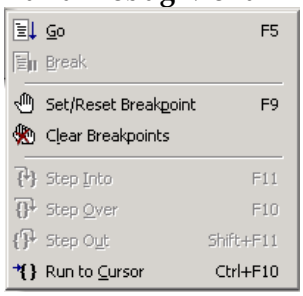
Button/List	Function
	Rebuild a progran
	Build a program.
	Compile a program.
	Execute a program.
	Activate the debugger.
	Break.
	Kill the program.
	Insert/Remove (toggle) breakpoint.
	Remove all breakpoints.
	Step into.
	Step over.
	Step out.
	Run to cursor.
	Display/Hide (toggle) Watch window.
	Display/Hide (toggle) Stack window.
	Program limits.
	Recordable variables.

Table 3-2: ***Build Toolbar Elements***

3.1.3 The Menu Bar

The menu bar along the top of the Program Editor provides access to the full range of tools and options. The main menu options are described in Table 3-3.

Menu	Option	Sub-option	Description
File 			Standard Windows options for opening, saving and manipulating program files, along with options to upload programs from and save programs to the connected drive.
Edit 			Standard Windows Undo/Redo, Cut-and-Paste, and Find/Find All options.
View 	Toolbars		Display/Hide Standard, Communication Build and Workspace toolbars.
	Windows		Display/Hide Workspace, Output, Watch, Stack and Communication Info windows.
View-Toolbars 	Status Bar		Display/Hide status bar at the bottom.
	Active Line		Jump to and highlight Previous or Next active line of code.
View-Windows 			

Menu	Option	Sub-option	Description
Build 	Rebuild		Recompile and download program.
	Build		Compile and download program.
	Compile		Compile a program source into an executable code.
	Kill Program		Stop program execution.
	Execute		Run program.
Build-Debug Menu 	Debug	Go	Run debugger.
		Break	Halt (suspend) program execution and return control to debugger.
		Set/Reset Breakpoint	Select/Cancel line for breakpoint.
		Clear Breakpoints	Delete all selected breakpoints.
		Step Into	Enter function and stop at first command.
		Step Over	Execute the next instruction line and then halt.
		Step Out	Complete the current function and then step out to the location immediately following the line on which the function was called.
		Run to Cursor	Halt execution at the instruction line at which the cursor is standing.

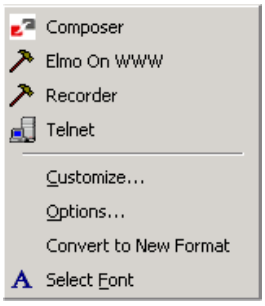
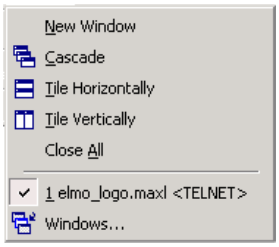
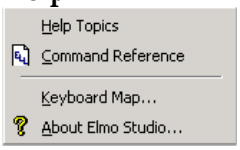
Menu	Option	Sub-option	Description
Tools 	Composer		Open the Elmo Composer.
	Elmo on WWW		Open browser on Elmo's web site
	Recorder		Open the Recorder
	Telnet		Open a Telnet connection
	Customize		Display Customize dialog box for altering menu, toolbar and keyboard options.
	Options		Display Options dialog box to select Debug and Build parameters.
	Convert to New Format		Convert a program coded in Elmo .ell format to Elmo .ehl format.
	Select Font		
Window 	New Window		Open a new program window.
	Cascade		Lay out windows in a cascade
	Tile Horizontally		Lay out windows horizontally
	Tile Vertically		Lay out windows vertically
	Close All		Close all windows
	Windows		Manipulate all open windows.
Help 	Help Topics		Display Elmo Studio online Help.
	Command Reference		Open the Command Reference
	Keyboard Map		Display a list of menu options, their accelerator key combinations and their descriptions.
	About Elmo Studio		Display information about the currently installed Elmo Studio version.

Table 3-3: ***Menu Bar Options***

3.1.4 Workspace Menus (accessed with right mouse button)

When working in the Workspace, Workspace-specific menus are available. These are described in Table 3-4.

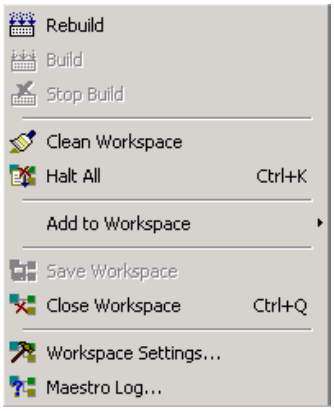
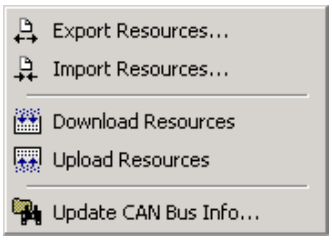
Menu	Option	Sub-option	Description
File View Menu 	Rebuild		Recompile and download program.
	Build		Compile and download program
	Stop Build		
	Clean Workspace		
	Halt All		
	Add to Workspace		
	Save Workspace		
	Close Workspace		
	Workspace Settings		
	Maestro Log		
Resource View Menu 	Export Resources		
	Import Resources		
	Download Resources		
	Upload Resources		
	Update CAN Bus Info		

Table 3-4: ***Workspace Menu Options***

3.1.5 Resource Editing Menus (right mouse button only)

When adding new resources or editing existing ones, a few additional menu options are available, namely Insert, Remove and Properties. These are described in Table 3-5.

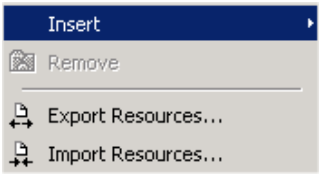
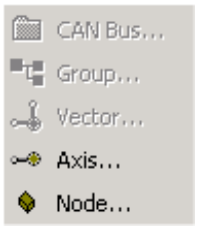
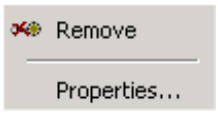
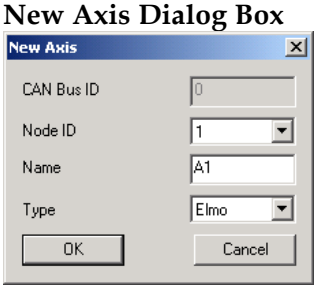
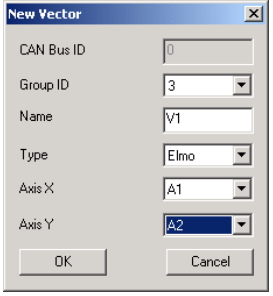
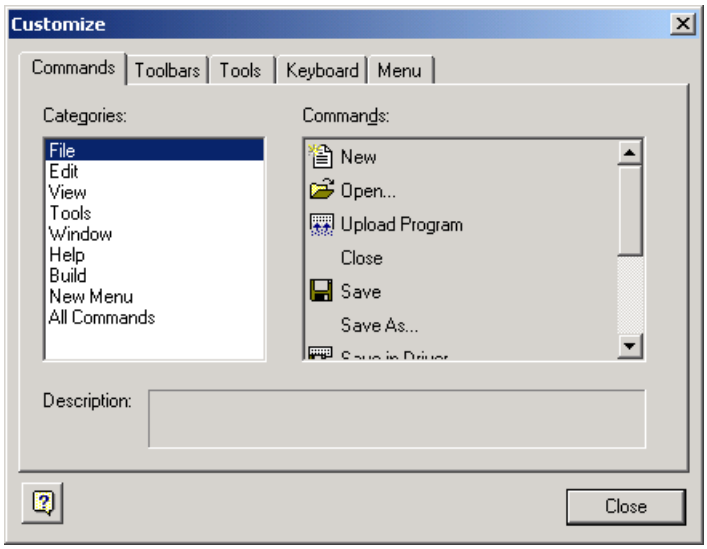
Menu	Option	Sub-option	Description
Resource View Menu 	Insert		Insert Node, Axis, Vector, Group or CAN bus.
	Remove		Remove a resource. This must be done in hierarchical order. For example, vectors must be removed before the axes with which they are they are made.
Insert Resource Menu 		CAN Bus	
		Group	
		Vector	
		Axis	
		Node	
Existing Resource Menu 		Remove	
		Properties	
New Axis Dialog Box 		Node ID	Select Node ID from those available
		Axis Name	User defined
		Type	Elmo or CANopen
	Note: This is the same dialog box that opens when clicking the properties button when highlighting an axis.		
New Vector Dialog Box 		Node ID	Select Node ID from those available
		Axis Name	User defined
		Type	Elmo or CANopen
	Note: This is the same dialog box that opens when clicking the properties button when highlighting an vector.		

Table 3-5: ***Resource Editing Menu Options***

3.1.6 Customizing the Elmo Studio

The **Tools** menu contains two options for customizing your Elmo Studio application to your mode of work: the Customize dialog boxes and the Options dialog boxes.

3.1.6.1 The Customize Dialog Boxes



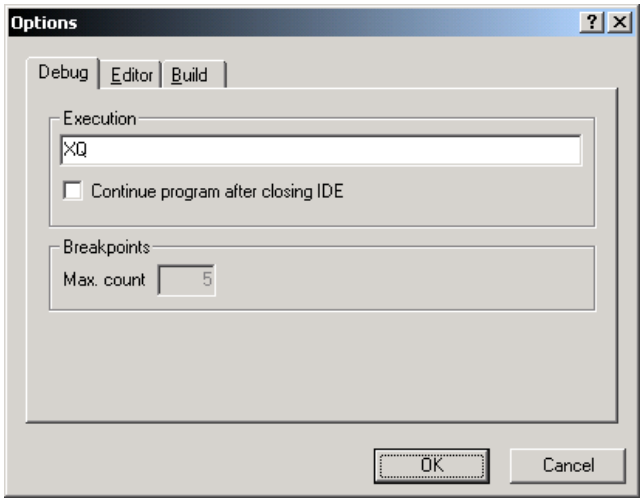
The following tabbed dialog boxes are available:

- Commands
  Enables you to add command buttons to the toolbar (by drag-and-drop) and to remove unneeded ones by dragging them off the toolbar.
- Toolbars
  For selecting which toolbars should be displayed or hidden, and to manipulate them as needed.
- Tools
  For adding and organizing frequently needed external applications to the Elmo Studio Tools menu.
- Keyboard
  For programming key combinations (“accelerators”) for frequently-used menu options.
- Menu
  For customizing the appearance of the various menu bars and popup menus.

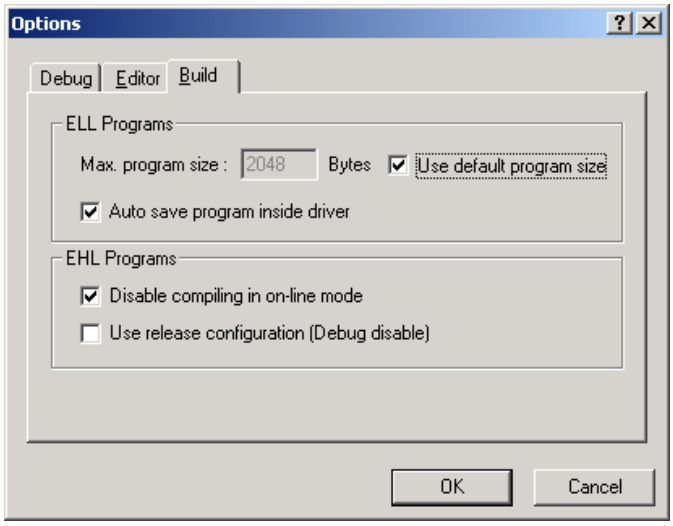
3.1.6.2 The Options Dialog Boxes

The following tabbed dialog boxes are available:

- Debug
  Enables you to select a command code for automatic debugging. Use the **Continue program after closing IDE** option to have the debug program continue to run even after the IDE has been closed.



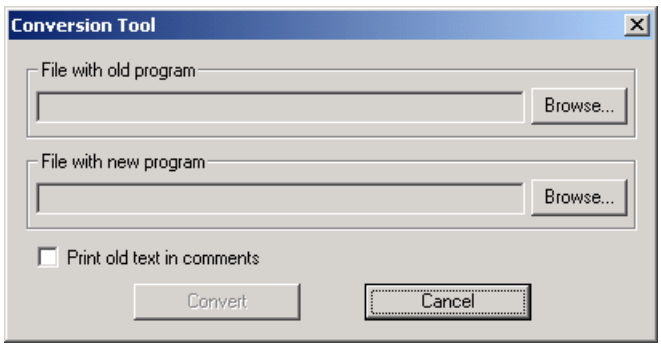
- Editor
  For displaying/hiding the selection margin and the numbers margin. The selection margin is the gray column to the left of the program text, which enables you to select the entire line adjacent to the cursor click position. The numbers margin adds a column of line numbers to the left of the program text.
- Build



- For defining parameters of new and "old" programs. For .ell programs, the maximum program size is displayed, along with the option to use it as the default program size. Use the **Auto save program inside driver** option to automatically save the program in the drive memory. The **Disable compiling in on-line mode** option disables the independent **Build - Compile** menu option so that compiling is performed only as part of the Build function.

3.1.6.3 Convert to New Format

You can use the **Tools - Convert to New Format** option to save a program, coded in earlier Elmo .ell format, in Elmo .ehl format. The Conversion Tool dialog box is used to select the existing (.ell) file and save it under a new name, in the new .ehl format.



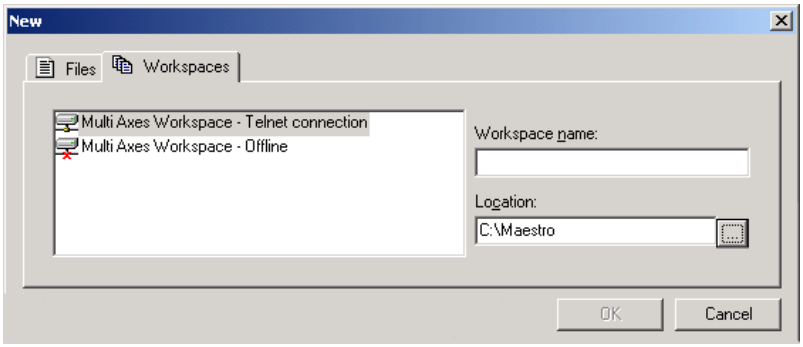
1. Click the **Browse** button next to the **File with old program** text box and navigate to the .ell file to be converted.
2. Click the **Browse** button next to the **File with new program** text box and navigate to the location at which the new file should be saved, giving it a new name if needed.
3. Click the **Print old text in comments** option if you wish to have the original text displayed, as comments, at the start of the new file.
4. Click **Convert** to activate the conversion process.

Chapter 4: Using the Program Editor

This chapter describes how to use the Program Editor.

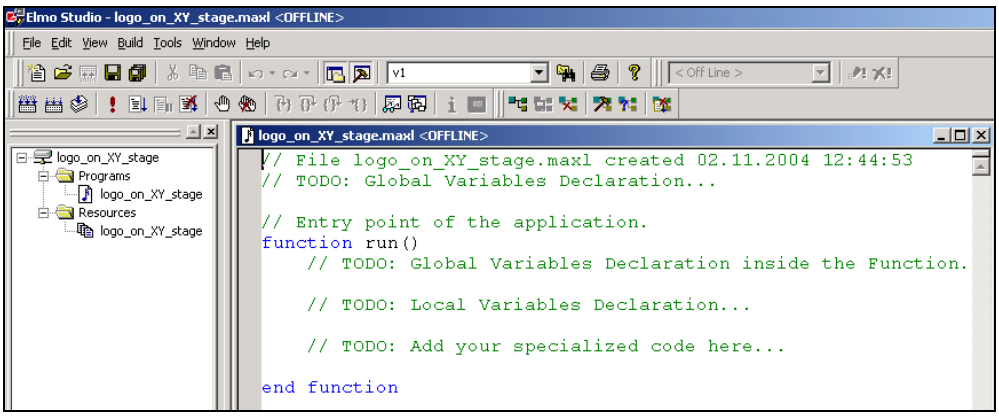
4.1 Creating a Workspace

To write a new program for subsequent download to a drive, click or select **File - New**. The New dialog box will be displayed:



1. Choose to work on-line via Telnet or off-line.
2. Give the workspace a name.
3. Choose a location to store the program files. Click on to browse with File Explorer.
4. Click **OK**. A Select Maestro window will open.

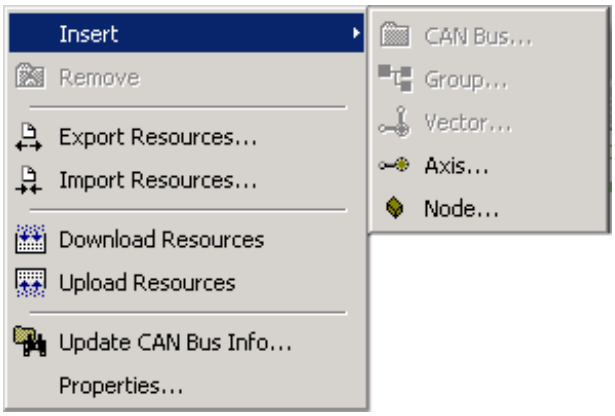
5. Select from the list of the Maestro's currently connected to the network.
6. A program template will open on the screen:



4.2 Setting up the Resources

Once a worksheet is open, the resources must be defined.:

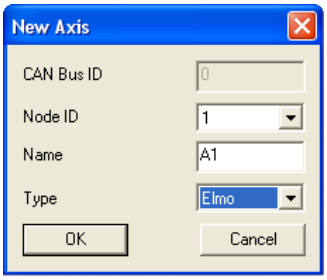
1. In the Resource view, highlight any line and press the right mouse button.
   The following menu opens:



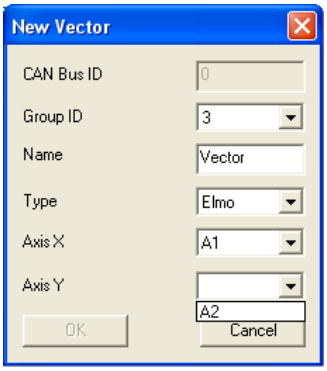
2. Click on the Insert option to get to the resource sub-menu.

In the Resource sub-menu start by defining the axes.

3. Select Axis and fill in the New Axis dialog box.
   Do this for as each axis on the network.



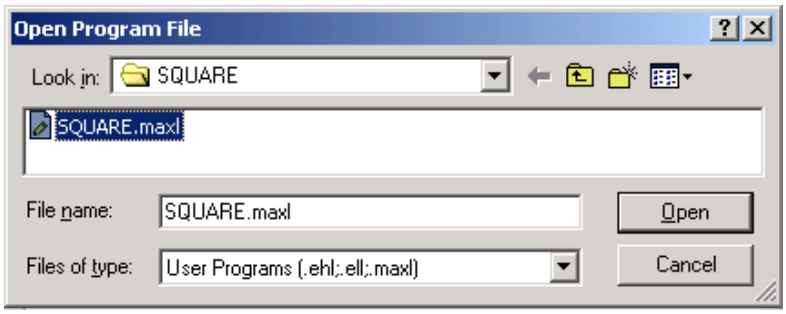
4. Select Vector and fill in the New Vector dialog box.
   Do this for as each vector on the network.

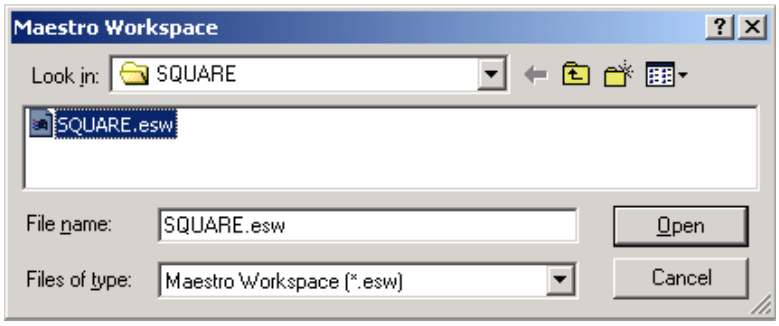


Once the resources have been defined they must be loaded on the *Maestro*.

5. Select the Download Resources command to load the resources on the *Maestro*.
6. Save the resource file to disc by selecting the Export Resources command.

4.3 Opening an Existing Program





4.4 Editing a Program File

To open an existing program file that resides on your computer, click or select **File - Open**. A program window will be opened with the selected file.

Once the file is open, you can edit the program as you would in any text editor, using the tools available in the Standard toolbar. You can perform regular searches using the **Edit - Find** function, and you can search for multiple occurrences of an item using the **Edit - Find All** option (or clicking).

When using the **Find All** option, the results of the search will, by default, be displayed in the Find in Files 1 tab of the Output window. To perform subsequent searches without overwriting the results of a previous search, select the **Output to pane 2** check box in the Find All dialog box.

4.5 Compiling a Program

Once your program is completed to your satisfaction, you can compile it. You may compile the program either online (while communication with the driver is active) or off-line. Alternatively, you may use the **Build** option to compile and download the program in one step (refer to section 4.6).

To compile a program, click or select **Build - Compile**. The Program Editor will compile the program and display all processing messages in the Build tab of the Output window.

When error messages occur during program execution and the program source needs to be fixed, you can double-click on the error message to locate the error in your program. When a Build operation finishes successfully with no errors, you can then execute the program (section 4.7) or run it using the Debug option (section 4.8).

4.6 Building a Program

The **Build** option enables you to have the Program Editor compile the program and automatically download it to the connected drive. To build a program, click or select **Build - Build**. The Program Editor will first compile the program. If no errors occur, it will then send the program to the Servo drive where it will be saved in the drive memory. If errors *do* occur during compiling, the Program Editor will record the errors and enter pointers in the program at locations where the errors occurred. It will then halt and will not download the program.

4.7 Running a Program

To run a program after it has been downloaded to a drive, click or select **Build - Execute**. The program will run independently of all debug options (such as Breakpoint and Step-by-step).

To stop a program while it is running, click or select **Build - Kill Program**.

4.8 Debugging

The Program Editor contains powerful tools for debugging the programs that you create and edit. It enables you to mark your program with breakpoints, and to control the debugging process according to your needs. Basically, you perform debugging according to the following steps:

1. Identify the section of the program where you suspect that a problem lies. This may be according to run-time error messages that you receive from the drive.
2. Mark the first instruction of that section with a breakpoint, clicking anywhere in the line and then clicking , pressing **<F9>** or selecting **Build - Debug - Set/Reset Breakpoint**. (You can cancel a breakpoint by repeating this action.)
3. You may also drag-and-drop variables from the program into the Watch window at the bottom. These are variables whose values you wish to know each time the program is suspended.
4. Start the debugging operation by clicking or selecting **Build - Debug - Go**. The debugger will execute the program until it reaches the first breakpoint, at which time program execution is halted. Each time the program you are debugging stops at a breakpoint, the debugger will update the Debug tag of the Output window with the relevant progress message. It will also indicate — with a yellow arrow and red highlight — the line of code at which the program stopped. Functions not yet returned will be displayed in the Stack window.
5. From here, you can use the relevant toolbar buttons or **Build - Debug** options (Table 4-1) to step through the program and continue the debugging operation manually.

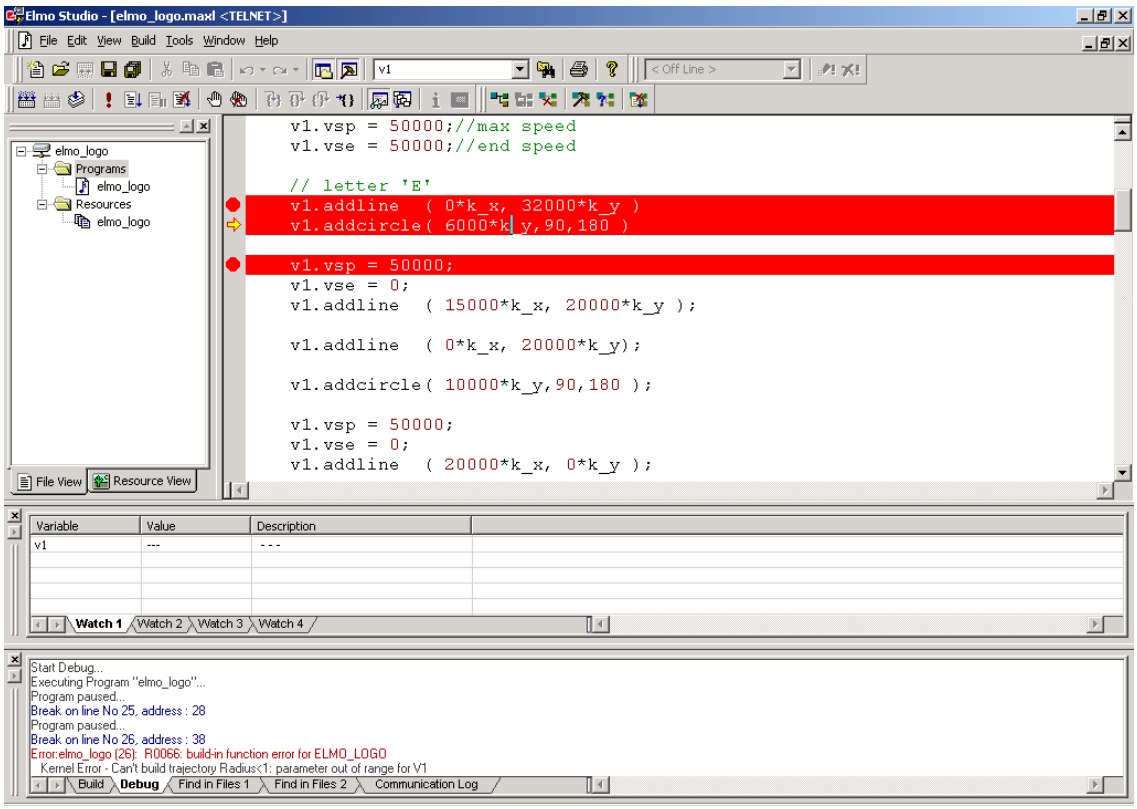


*Figure 4-1: **Debugging Process***

The following **debugging tools** are available for enabling you to manually debug your program in conjunction with the Program Editor debugger:

Button	Menu Option	Description
	Build - Debug - Break	Stop the debugger as it is running.
	Build - Debug - Step Into	Enter function and stop at first command.
	Build - Debug - Step Over	Execute next instruction and then stop.
	Build - Debug - Step Out	Continue and then stop at first instruction before current function is called.
	Build - Debug - Run to Cursor	Halt execution at source line at which the cursor is presently standing (no breakpoint needed).

Table 4-1: ***Debugging Tools***

Chapter 5: Host Communications

This chapter explains how to set up the Workspace to work with a specific Maestro as a host.

5.1 Setting Up the Host

A Select Maestro window pops up when starting a new worksheet. The window contains a list of Maestros currently attached to the network. The IP Address of the Maestro and its name are listed. Select the Maestro you plan to work with and click OK.

Figure 5-1 The Maestro Selection Window

5.2 Verifying or Changing the Host

To verify that you have set up the correct Maestro as the host, or to change to another Maestro, move the cursor into the File Viewer and click on the right mouse button. When the menu pops up select Workspace Settings. This will cause the Workspace Settings window to open. If the IP Address is wrong, click the Find button to open the Select Maestro window.

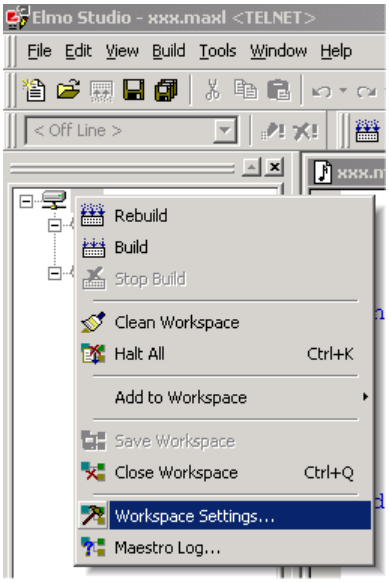


Figure 5-2 Right Mouse Button Selection Options

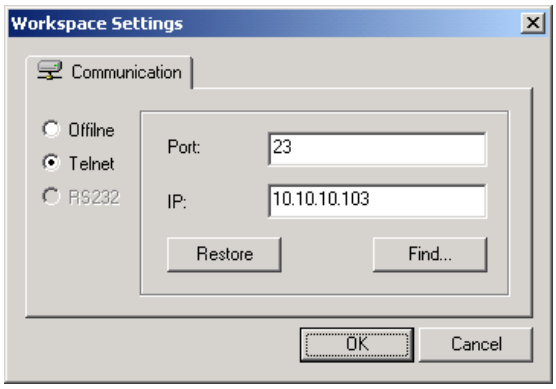


Figure 5-3 The Workspace Settings Window

Chapter 6: Command Line Interface

This chapter describes the *Maestro* commands that are available with its Command Line Interpreter (CLI).

6.1 Accessing the CLI

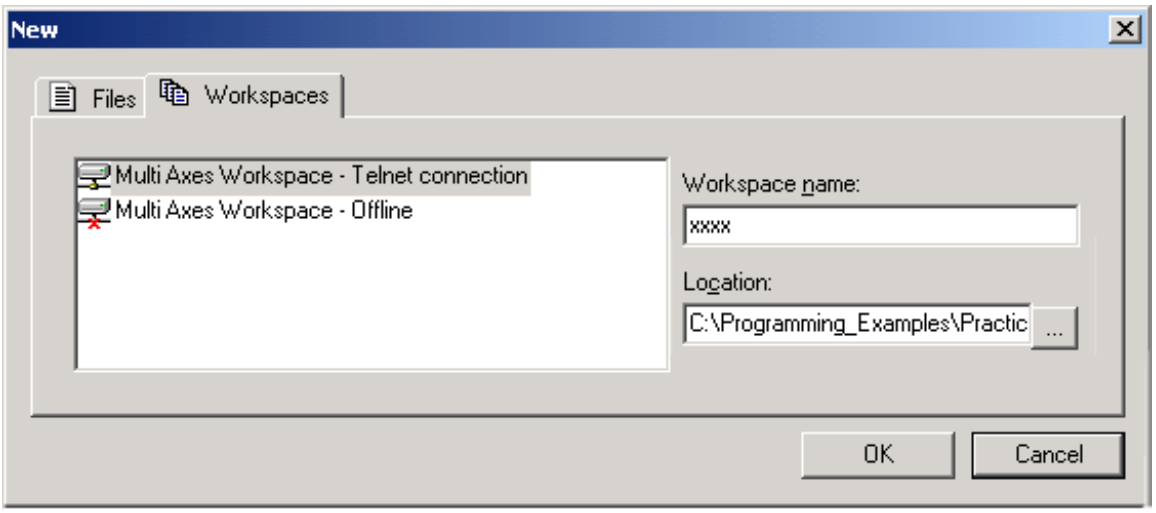


Figure 6-1 Opening a Workspace with a Telnet Connection

Figure 6-2 Selecting a *Maestro* on the Network

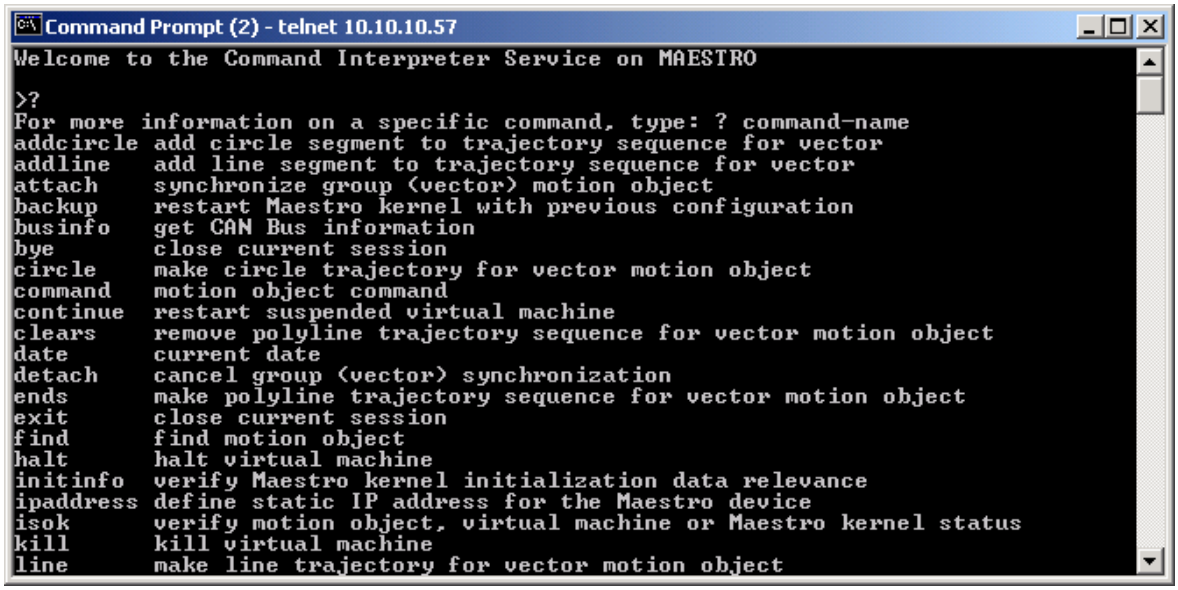


Figure 6-3 The *Maestro's* CLI Commands

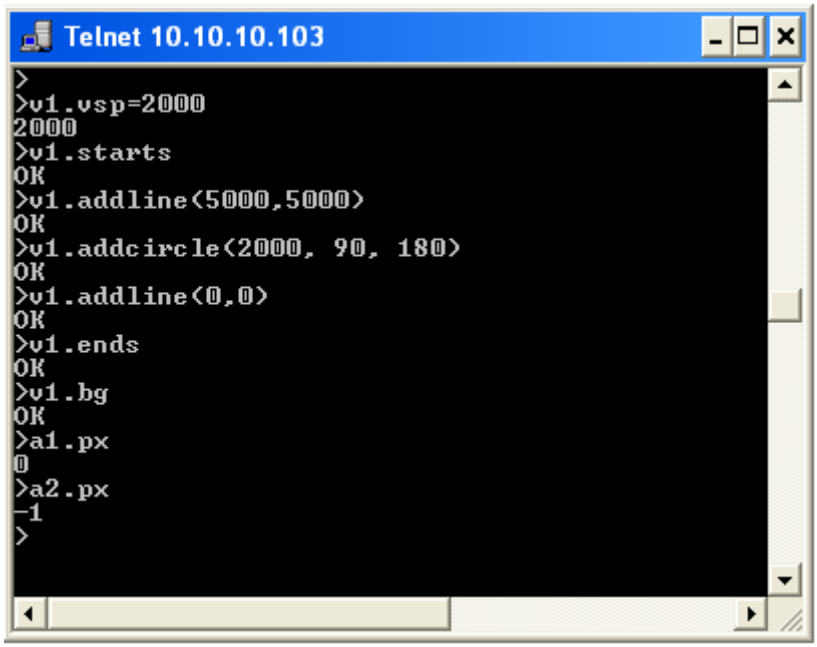


Figure 6-4 The *Maestro's* Command Line Interpreter

These commands characterize the reference command to the drive, or how the command reference should be specified.

Command	Description	Example & Parameters
addcircle	add circle segment to vector trajectory sequence	vector.addcircle(rad, alpha, delta)
addline	add line segment to vector trajectory sequence	vector.addline(x, y)
attach	synchronize group/vector motion object	vector.attach
backup	restart Maestro kernel with previous configuration	
businfo	get CAN Bus information	businfo 0 parameter: CAN Bus ID
bye	close current session	
circle	create circle trajectory for vector motion object	vector.circle(rad, alpha, delta)
command	motion object command: it is possible to send commands for node, axis, vector or group; length of command - 2 characters	axis.mo=1 get command format: obj.** or obj.**[n] set command obj.**=x or obj.**[n]=x
continue	restart suspended virtual machine	vm.continue
clears	remove polyline trajectory sequence for vector motion object	vector.clears
date	current date	
detach	cancel group/vector synchronization	vector.detach
ends	create polyline trajectory sequence for vector motion object	vector.ends
exit	close current session	
find	find motion object	find vector1 parameter: motion object
halt	halt virtual machine	vm.halt
initinfo	verify relevance of Maestro kernel initialization data	
ipaddress	define static IP address for the Maestro format: ipaddress - to display IP address ipaddress *.*.*.* - to define address	ipaddress 10.10.10.96
isok	verify motion object, virtual machine or Maestro kernel status	vm.isok
kill	kill virtual machine	vm.kill

Command	Description	Example & Parameters
line	make line trajectory for vector motion object	vector.line(x, y)
list	get motion object list parameters target object type; can be a or b: a) motion objects type: node, i/o, axis, vector, group b) virtual machines type: **vmf** - all non-initialized machines **vm** - all initialized machines **vmr** - all running machines **vma** - all aborted machines	list b:0 axis parameter: CAN Bus ID
netmask	define Subnet Mask for static IP address of the Maestro netmask - to display Subnet Mask netmask *.*.*.* - to define Subnet Mask	netmask 255.255.255.0
pause	suspend virtual machine	vm.pause
property	motion object property: possible to set/get property for axis, vector or group length of property name - 3 characters first character of property name: a for axis v for vector g for group, format: obj.*** or obj.***=x a) axis properties: **abl** - PVT buffer length (default value: 64) **ame** - motion events listener mask (def. val.: 0) v) vector properties: **vac** - trajectory acceleration (counts per second^2) **vbl** - PVT buffer length (default value: 32) **vcr** - const PVT resolution flag (default value: 0) **vdc** - trajectory deceleration (counts per second^2) **vln** - number of trajectory cycles (default value: 1) **vqt** - trajectory step time (millisecond), **vsc** - smoothing curve mode: **0**-non-smoothed **1**-smoothed with max. velocity (default value) **2**-smoothed with fixed radius **vse** - trajectory end velocity (counts per second), **vsf** - smooth factor (time in ms for max. speed access) **vsm** - send command mode: **0**-sync mode - use group id **1**-async mode - use node id (default value), **vsp** - trajectory fixed velocity (counts per second) **vsr** - radius (counts) for smoothing curve mode **vsc**=2 **vtt** - trajectory fixed time (millisecond) **vum** - trajectory build mode: **1**-max velocity mode **2**-fixed time mode **3**-fixed velocity mode (default value)	

Command	Description	Example & Parameters
	g) group properties: **gsm** - send command mode: **0**-sync mode - use group id **1**-async mode - use node id (default value)	grp.gsm=1
quit	close current session	
restart	restart Maestro kernel with current configuration	
run	run virtual machine	vm.run
starts	begin polyline trajectory sequence for vector motion object	vector.starts
subnet	define Subnet Address for the Maestro format: subnet - to display Subnet Address subnet *.*.*.* - to define Subnet Addr.	subnet 10.10.10.0
sync	define sync parameters format: sync(id, timeout) **id** - CAN Bus ID **timeout** - sync sending period (0 – for cancel)	
time	current time	
trj	set trajectory for axis **axis.trj** - to display current trajectory **axis.trj**(trjname) - set current trajectory **axis.trj**(trjname, n) - set current trajectory and loop number (-1 : infinitely)	a1.trj(moshe,3)
ver	Maestro Version	

Chapter 7: Motion Instructions

This chapter takes a look at the organization of *Maestro* motion object programming. The Maestro contains the following set of **motion objects**:

- **Axis** - the most basic *Maestro* motion object - used to control the motion of a single motor/axis
- **Group** - composite *Maestro* motion object, which is made of two or more *Maestro* motion objects of the same type. This object can be used to synchronize the operation of its "members".
- **Vector** - a particular case of Group, which is made of two Axes of the same type. This object can be set up as a trajectory.

All motion objects use the same set of **motion instructions** which include:

- **commands –** device control messages (these messages are similar to the ones used by *SimplIQ* drives and are described in the *SimplIQ Command Reference*)
- **properties** - determines *Maestro* motion object behavior
- **functions –** predefined set of motion functions

Motion instructions can be sent from a terminal or from a Maestro user program.

7.1 Axis

7.1.1 Axis Motion Commands

Motion Commands are described in detail in the *SimplIQ Command Reference Manual*. When used to program the Maestro, their syntax is as follows:

- To send commands for set parameter value:
  <Axis name>.Command=<Value>
- To receive parameter values:
  <Value>=<Axis name>.Command

For example:

```
A1.UM = 2;
A1.MO = 1;
A2.BG;
A2.ST.
```

7.1.2 Axis Properties

Motion properties apply to Maestro programs only. Axis related properties are listed below:

- To set property values:
  <Axis name>.Property=<Value>
- To get property values:
  <Value>=<Axis name>. Property

Property:	**ADT - dotrj** command delay
Dimension:	Milliseconds
Default:	20
Limitation:	20-254
Type:	Unsigned int

Property:	**AFP –** first points portion size for PVT trajectory
Dimension:	
Default:	5
Limitation:	3 – 10
Type:	Unsigned int

Property:	**AME –** motion events listener mask
Note:	Available events: MOTION_COMPLETE = 0x01, MAIN_HOMING_COMPLETE = 0x02, AUX_HOMING_COMPLETE = 0x04, MOTOR_SHUTDOWN = 0x08, MOTOR_STARTED = 0x10, USER_PROGRAM_EMIT = 0x20, DIGITAL_INPUT_EVENT = 0x80,
Default:	0x01 (MOTION_COMPLETE |)
Limitation:	0 – 254
Type:	Unsigned int

Property:	**APP –** number of PVT points in portion of PVT trajectory
Dimension:	
Default:	5
Limitation:	3 – 10
Type:	Unsigned int

7.1.3 Axis Functions

Motion functions apply to Maestro programs only. Axis related functions are listed below:

- To call a function:
  <Return Value>=<Axis name>.Function[([Parameters])]

Function:	**trj** – used for initialize the PVT mechanism and loading a PVT trajectory table to an axis
Call Format:	<Axis name>.trj(<Trajectory name>) <Axis name>.trj(<Trajectory name>, <Loop number>)
Parameters:	<Trajectory name> - name of PVT table (trajectory) <Loop number> - number of PVT table (trajectory) loops, if 0 - infinite
Return Value	Terminal only OK: Trajectory name FAILED: Error message
Example	A1.trj(table, 2); A1.bg for axis name A1 run trajectory name "table " 2 - times

Function:	**dotrj** – used for initialize PVT mechanism and load initial portion of the PVT trajectory table to an axis and immediately obtain command 'BG' – begin motion.
Call Format:	<Axis name>.dotrj(<Trajectory name>) <Axis name>.dotrj(<Trajectory name>, <Loop number>)
Parameters:	<Trajectory name> - name of PVT table (trajectory) <Loop number> - number of PVT table (trajectory) loops, if 0 - infinite
Return Value	Terminal only OK: Trajectory name FAILED: Error message
Example	A1.dotrj(table) for axis name A1 run trajectory name "table "

Function:	**isok** – to verify motion object status
Call Format:	<Axis name>.isok
Parameters:	
Return Value	Terminal only OK: Ok FAILED: Error status
Limitation:	Terminal only
Example	A2.isok

Function:	**startp -** begin user's PVT trajectory sequence for an Axis
Call Format:	< Axis name>.startp() < Axis name>.startp(<Trajectory name >)
Parameters:	<Trajectory name> - name of Axis PVT table (trajectory) to save
Example	a1.startp(mytable) – begin user's PVT trajectory named "mytable" For more information see the ends function example

Function:	**addpoint -** add PVT point to user's trajectory sequence for Axis which began with a starts function
Call Format:	< Axis_name>.addpoint(<Pos >,<Vel >, <T>)
Parameters:	<Pos > - position for PVT point (counts), <Vel >, - velocity for PVT point (counts per second), <T> - interpolation period for PVT point (milliseconds)
Return Value	Terminal only OK: Ok FAILED: Error message
Example	a1.addpoint(1000, 30000, 10) For more information see the ends function example

Function:	**ends -** make user's trajectory sequence for Axis motion object
Call Format:	< Axis_name>.ends()
Parameters:	
Return Value	Terminal only OK: Ok FAILED: Error message
Example	a1.startp(mytable) the trajectory name mytable a1.addpoint(10, 1000, 10) a1.addpoint(1000, 30000, 10) ……. a1.addpoint(5000, 1000, 10) a1.ends() after this the trajectory named "mytable" will contain the user's PVT table

Function:	**error –** to get motion object last error
Call Format:	<Axis name>.isok
Return Value	Terminal only OK: Ok () FAILED: Error message
Limitation:	Terminal only
Example	A2.error

7.2 Group

7.2.1 Group Motion Commands

Group motion commands can be sent synchronously or asynchronously - *simultaneously* to a Group with a PDO ID, or *serially* to all members of a group according **GSM property value**. However, only the asynchronous parcels sent serially to all members of a group provides a reliable check of operating performance.

- To send set parameter values:
  <Group name>.Command[=<Value>]

For example:

G1.PX = 0;
G1.MO = 1;
G2.BG;
G2.ST.

7.2.2 Group Properties

- To set property values:
  < Group name>.Property=<Value>
- To get property values:
  <Value>=< Group name>. Property

Property:	**GBT** - synchronized start command delay
Dimension:	Milliseconds
Default:	3
Limitation:	2-254
Type:	Unsigned int

Property:	**GME** – motion events listener mask
Note:	Available events: MOTION_COMPLETE = 0x01, MAIN_HOMING_COMPLETE = 0x02, AUX_HOMING_COMPLETE = 0x04, MOTOR_SHUTDOWN = 0x08, MOTOR_STARTED = 0x10, USER_PROGRAM_EMIT = 0x20, DIGITAL_INPUT_EVENT = 0x80,
Default:	0x01 (MOTION_COMPLETE)
Limitation:	0 – 254
Type:	Unsigned int

Property:	**GSM** – send group command mode
Note:	Available: 0 – synchronized mode, use Group ID 1 – asynchronized mode, use Node ID of members
Default:	0
Limitation:	0,1
Type:	Unsigned int

7.2.3 Group Functions

- To call function:
  <Return Value>=<Group name>.Function[([Parameters])]

Function:	**attach** – used to synchronize all members of the Group according the resource file. The attach function must be called before the first group operation.
Call Format:	<Group name>.attach()
Parameters:	
Return Value	Terminal only OK: Ok FAILED: Error message
Example	G1.attach()

Function:	**detach** – cancel Group synchronization. Group operation calls, after detach, cause a RUNTIME error.
Call Format:	<Group name>.detach()
Parameters:	
Return Value	Terminal only OK: Ok FAILED: Error message
Example	G1.detach()

Function:	**isok** – to verify motion object status
Call Format:	<Group name>.isok
Parameters:	
Return Value	Terminal only OK: Ok FAILED: Error status
Limitation:	Terminal only

Example	Grp.isok
Function:	**error –** to get motion object last error
Call Format:	<Group name>.isok
Parameters:	
Return Value	Terminal only OK: Ok FAILED: Error message
Limitation:	Terminal only
Example	Grp.error

7.3 Vector

7.3.1 Vector Motion Commands

Vector commands can be sent synchronously or asynchronously according to **VSM** value - *simultaneously* to a Vector with a PDO ID or *serially* to all members of a group. However, only the asynchronous parcel sent serially to all members of a group provides a reliable check of operating performance results.

A vector is a set of axes that provide two dimension interpolated motion.

When a vector is active, the member axes can receive commands not only from it, but also from Axis instructions.

A Vector cannot be activated if one of its member Axes is a member of another active Vector.

- To send a set parameter value:
  < Vector name>.Command[=<Value>]

For example:

```
V1.AC = V1.DC=1000000;
V1.MO = 0;
V2.BG;
V2.ST.
```

7.3.2 Vector Properties

- To set a property value:
  < Vector name>.Property=<Value>
- To get a property value:
  <Value>=< Vector name>. Property

Property:	**VAC** - trajectory acceleration
Dimension:	Counts per second square
Default:	28000000
Limitation:	
Type:	Float

Property:	**VAE** - calculatedacceleration error for PVT trajectory
Dimension:	%
Default:	0.005
Limitation:	0-1
Type:	Float

Property:	**VPE** – calculatedposition error for PVT trajectory
Dimension:	%
Default:	0.005
Limitation:	0-1
Type:	Float

Property:	**VBT** - synchronized start command delay
Dimension:	Milliseconds
Default:	3
Limitation:	2-254
Type:	Unsigned int

Property:	**VCR** - const PVT table resolution flag
Note:	1 - const PVT resolution 0 – various PVT resolution
Default:	0
Limitation:	0, 1
Type:	Unsigned int

Property:	**VDC** - trajectory deceleration
Dimension:	counts per second square
Default:	28000000
Limitation:	
Type:	Float

Property:	**VDT –** defines delay for **dotrj** command
Dimension:	Milliseconds
Default:	30
Limitation:	20-254
Type:	Unsigned int

Property:	**VFP –**number of points downloaded in the first portion of a PVT trajectory
Dimension:	
Default:	5
Limitation:	3 – 10
Type:	Unsigned int

Property:	**VLN –** number of trajectory cycles for a Vector
Note:	0 - infinite
Default:	1
Limitation:	0, 1 … N
Type:	Unsigned int

Property:	**VME –** motion events listener mask
Note:	Available events: MOTION_COMPLETE = 0x01, MAIN_HOMING_COMPLETE = 0x02, AUX_HOMING_COMPLETE = 0x04, MOTOR_SHUTDOWN = 0x08, MOTOR_STARTED = 0x10, USER_PROGRAM_EMIT = 0x20, DIGITAL_INPUT_EVENT = 0x80,
Default:	0x01 MOTION_COMPLETE
Limitation:	0 – 254
Type:	Unsigned int

Property:	**VNT** – calculated PVT trajectory min step time
Dimension:	Milliseconds
Default:	2
Limitation:	1–254
Type:	Unsigned int

Property:	**VXT** – calculated PVT trajectory max step time
Dimension:	Milliseconds
Default:	7
Limitation:	1–254
Type:	Unsigned int

Property:	**VPP** – single points portion size for PVT trajectory
Dimension:	
Default:	5
Limitation:	3 – 10
Type:	Unsigned int

Property:	**VSC** – PVT trajectory smoothing curve mode
Note:	Available: 0-non smoothed 1-smoothed with max velocity 2-smoothed with fixed radius 3-smoothed with fixed dist from the corner
Default:	0
Limitation:	1-254
Type:	Unsigned int

Property:	**VSD** – PVT trajectory smooth distance from the corner point
Dimension:	Counts
Default:	0
Limitation:	
Type:	Unsigned int

Property:	**VSE** – PVT trajectory end velocity
Dimension:	Counts per second
Default:	0
Limitation:	
Type:	Float

Property:	**VSF** – smooth factor
Dimension:	time in millisecond for max speed access
Default:	0
Limitation:	
Type:	Float

Property:	**VSM** – send vector command mode
Note:	Available: 0 – synchronized mode, use Group ID of Vector 1 – asynchronized mode, use Node ID of members
Default:	0
Limitation:	0,1
Type:	Unsigned int

Property:	**VSP** – PVT trajectory fixed (profile) velocity
Dimension:	Counts per second
Default:	0
Dependencies:	In max velocity mode (vum = 1) vsp = the limit of max velocity value In fixed time mode (vum = 2) vsp = the limit of max velocity value In fixed velocity mode (vum = 3) vsp = fixed velocity value
Type:	Float

Property:	**VSR** – PVT trajectory smooth radius (curve mode VSC=2)
Dimension:	Counts
Default:	0
Limitation:	
Type:	Unsigned int

Property:	**VTT** – PVT trajectory fixed time
Dimension:	Milliseconds
Default:	0
Limitation:	
Type:	Unsigned int

Property:	**VUM** – PVT trajectory build mode
Note:	Available: 1 – max velocity mode 2 - fixed time mode 3 - fixed velocity mode
Default:	3
Limitation:	0-254
Type:	Unsigned int

Property:	**VVE** - calculation velocity error for PVT trajectory
Dimension:	%
Default:	0.005
Limitation:	0-1
Type:	Float

7.3.3 Vector Functions

- To call a function:
  <Return Value>=<Vector name>.Function[([Parameters])]

Function:	**attach** – used for synchronize all members of the Vector according the resource file. The attach function must be called before the first vector operation.
Call Format:	< Vector name>.attach()
Parameters:	
Return Value	Terminal only OK: Ok FAILED: Error message
Example	G1.attach()

Function:	**detach –** cancel Vector synchronization
Call Format:	< Vector name>.detach()
Parameters:	
Return Value	Terminal only OK: Ok FAILED: Error message
Example	V1.detach()

Function:	**isok –** to verify motion object status
Call Format:	<Vector name>.isok
Parameters:	
Return Value	Terminal only OK: Ok FAILED: Error status
Limitation:	Terminal only
Example	V.isok

Function:	**error –** to get motion object last error
Call Format:	<Vector name>.isok
Parameters:	
Return Value	Terminal only OK: Ok FAILED: Error message
Limitation:	Terminal only
Example	V.error

Function:	**trj –** used for initialize PVT mechanism and load Vector PVT trajectory table
Call Format:	<Vector name>.trj(<Trajectory name>) <Vector name>.trj(<Trajectory name>, <Loop number>)
Parameters:	<Trajectory name> - name of Vector PVT table (trajectory) <Loop number> - number of Vector PVT table (trajectory) loops, if 0 - infinite
Return Value	Terminal only OK: Trajectory name FAILED: Error message
Example	v1.trj(table, 2); v1.bg for vbector name v1 run trajectory name “table ” 2 - times

Function:	**dotrj –** used for initialize PVT mechanism and load Vector PVT trajectory table. After this the axes (members of the Vector) immediately obtain command 'BG' – begin motion.
Call Format:	<Vector name>.dotrj(<Trajectory name>) <Vector name>.dotrj(<Trajectory name>, <Loop number>)
Parameters:	<Trajectory name> - name of Vector PVT table (trajectory) <Loop number> - number of Vector PVT table (trajectory) loops, if 0 - infinite
Return Value	Terminal only OK: Trajectory name FAILED: Error message
Example	V1.dotrj(table) for vector name v1 run trajectory name "table "

Function:	**starts -** begin polyline trajectory calculation sequence for Vector
Call Format:	<Vector name>.starts() <Vector name>.starts(<Trajectory name >)
Parameters:	<Trajectory name> - name of Vector PVT table (trajectory) for saving
Return Value	Terminal only OK: Ok FAILED: Error message
Example	V1.starts() - begin polyline trajectory calculation without saving the results in afile V1.starts(traj_name) - begin polyline trajectory calculation with saving result in file trajectory named "traj_name" For more information see the vector ends function example

Function:	**startp -** begin user's PVT trajectory sequence for Vector
Call Format:	<Vector name>.startp() <Vector name>.startp(<Trajectory name >)
Parameters:	<Trajectory name> - name of Vector PVT table (trajectory) for saving
Return Value	Terminal only OK: Ok FAILED: Error message
Example	V1.startp() - begin user's PVT trajectory without saving result in file V1.startp(traj_name) - begin user's PVT trajectory with saving result in file trajectory named "traj_name" For more information see the vector ends function example

Function:	**clears -** remove user's or polyline trajectory sequence for Vector motion object
Call Format:	<Vector name>.clears()
Parameters:	
Return Value	Terminal only OK: Ok FAILED: Error message
Example	Vec1.clears()

Function:	**addcircle -** add circle segment to trajectory sequence for Vector motion object
Call Format:	<Vector name>.addcircle(int <Radius>,float <Start Angle>,float <Sweep Angle>)
Parameters:	int <Radius> - radius of circle segment (counts) float <Start Angle> - start angle of circle segment (grads) float <Sweep Angle> - sweep angle of circle segment (grads)
Return Value	Terminal only OK: Ok FAILED: Error message
Example	Vec1.addcircle(rad, alpha, delta) For more information see the vector ends function example.

Function:	**addline -** add linear segment to trajectory sequence for Vector motion object
Call Format:	<Vector name>.addline(int <Pos X>,int <Pos Y>)
Parameters:	int <Pos X>,<Pos Y> - destination position of linear segment (counts), beginning of segment – current position
Return Value	Terminal only OK: Ok FAILED: Error message
Example	Vec1.addline(1000, 2000) For more information see the vector ends function example.

Function:	**addpoint -** add PVT point to user's trajectory sequence for Vector
Call Format:	<Vector name>.addpoint(int <Pos X>,int <Vel X>,int <Pos Y>,int int <Vel Y>, int <T>)
Parameters:	int <Pos X>,<Pos Y> - position for PVT point (counts), int <Vel X>,<Vel Y> - velocity for PVT point (counts per second), int <T> - interpolation period for PVT point (milliseconds) 0 - 255
Return Value	Terminal only OK: Ok FAILED: Error message
Example	Vec1.addpoint(1000, 30000, 2000, 40000, 10) For more information see the vector ends function example.

Function:	**ends -** make user's or polyline trajectory sequence for Vector motion object
Call Format:	<Vector name>.ends()
Parameters:	
Return Value	Terminal only OK: Ok FAILED: Error message
Example	v1.startp(mytable) - begin the trajectory name myable v1.addpoint(10, 1000, 20, 2000, 10) v1.addpoint(1000, 30000, 4000, 50000, 10) ……. v1.addpoint(5000, 1000, 4000, 2000, 10) v1.ends() - after this, the trajectory name "mytable" will contain the user's PVT table * * * v1.start(mytable) - begin the trajectory name myable v1.addline(1000,2000) v1.addcircle(1000, 180, -180) ……. v1.addline(5000, 1000) v1.ends() - after this the trajectory name "mytable" will contain the PVT table

Function:	**circle -** make circle trajectory for Vector motion object
Call Format:	<Vector name>.circle(int <Radius>,float<Start Angle>,float<Sweep Angle>)
Parameters:	int <Radius> - radius of circle trajectory (counts) float<Start Angle> - start angle of circle trajectory (grads) float<Sweep Angle> - sweep angle of circle trajectory (grads)
Return Value	Terminal only OK: Ok FAILED: Error message
Example	Vec1.circle(rad, alpha, delta);Vec1.bg

Function:	**line -** make linear trajectory for Vector motion object
Call Format:	<Vector name>.line(int <Pos X>, int <Pos Y>)
Parameters:	int <Pos X>,<Pos Y> - destination position of linear trajectory (counts), beginning of trajectory – current position
Return Value	Terminal only OK: Ok FAILED: Error message
Example	Vec1.line(1000, 2000);Vec1.bg

7.4 General Instructions

Function:	**sync** - begin sync CAN messages
Call Format:	sync (int <bus_numer>,float<sync_period>)
Parameters:	< bus_num > CAN bus number used to activate sending SYNC messages < sync_period > - SYNC period value in millisecond
Return Value	Terminal only OK: Ok FAILED: Error message
Example	sync(0,20) – for bus number 0 activate send sync per 20 milliseconds.

Function:	**tstamp** - set timestamp period
Call Format:	tstamp(int <bus_numer>,int<period>)
Parameters:	< bus_num > CAN bus number used to start sending SYNC messages <period> - period of timestamp sending in sync periods.
Return Value	Terminal only OK: Ok FAILED: Error message
Example	tstamp(0,5) – for bus number 0, after each 5 SYNC messages timestamp message will be send.

Function:	**List -** Get list of all Maestro kernel resource objects. An object is a node or group of nodes combined as vectors or other. object logic name: any string name. object CAN ID: any number 1-127 object CAN group ID: any number 1-127 (128 is not active) CAN Bus number: Bus 0,1… type of object: ElmoAxis, 402, 401, etc. profile: when object is axis, the profile it supports is reported. name: as in object 0x1008 (if exists) man: manufacturer as in object ### (if exists) members: the axes number that are grouped. Axis x:, Axis y: - axis names according to related coordinate of vector.
Call Format	list [[b:num] axis] [[b:num] vector] [[b:num] node] [vm] [vmf] [vmr] where: list: retrieves all information list [[b:num] axis]: retrieve motion axis information (Elmo or 402) on BUS b list [[b:num] vector]: retrieve information about the vector objects on BUS b. list [[b:num] node]: retrieve information about any non-motion node on BUS b list [vmf] [vm] [vmr]: retrieve the user programs names that are resident in the Maestro where: vmf: program exists in the Maestro flash vm: program virtual machine space is ready vmr: program is running
Parameters:	[[b:num] axis] [[b:num] vector] [[b:num] node] [vm] [vmf] [vmr]
Limitations	Terminal only
Example	list list axis list b:0 axis list b:1 vector b:1 node

Function:	**find -** Search for object according to the object's logical name. If an object exists, the information is displayed as a "list". If object does not exists the Maestro returns 'Object Not Defined"
Call Format:	find <object_name>
Limitations	Terminal only
Example	Find axis_1

Function:	**restart -** Restart the Maestro by: Closing all existing objects Kill all virtual machine Restart Maestro kernel Apply the sessions and objects according to the existing configuration file. Starts AUTOEXEC program if exists.
Call Format:	restart
Limitations	Terminal only

Chapter 8: MAXL Program Language

This chapter describes how to write Maestro programs on internal language - MAXL.

8.1 Lexical Conventions

This paragraph includes the fundamental elements of a MAXL (Multi Axis Language) program, as they are meaningful to the compiler. These elements, called "lexical elements" or "tokens", are used to construct statements, definitions, declarations, and so on, which are used to construct complete program. Tokens include:

- Comments
- Identifiers
- MAXL keywords
- Punctuations
- Operations

8.1.1 Comments

A comment is text that the compiler ignores but that is useful for programmers. Comments are normally used to annotate code for future reference. The compiler treats them as white space. You can use comments in testing to make certain lines of code inactive.

Chapter 8:A MAXL comment (like C++) is written in one of the following ways:

- The /* (slash, asterisk) characters, followed by any sequence of characters (including new lines), followed by the */ characters. This syntax is the same as ANSI C.
- The // (two slashes) characters, followed by any sequence of characters. A new line not immediately preceded by a backslash terminates this form of comment. Therefore, it is commonly called a "single-line comment." This syntax is the same as ANSI C++.

The comment characters (/*, */, and //) have no special meaning within a character constant, string literal, or comment. Comments using the first syntax, therefore, cannot be nested. Consider this example:

```
/* Intent:  Comment out this block of code.
   Problem: Nested comments on each line of code are illegal.
a = b*56  /* Initialize file string */
b=a/17; c=b*cos(alfa) /* Print status message */
*/
```

The preceding code will not compile because the compiler scans the input stream from the first /* to the first */ and considers it a comment. In this case, the first */ occurs at the end of the Initialize file string comment. The last */, then, is no longer paired with an opening /*.

Note that the single-line form (//) of a comment followed by end of line can have surprising effects. Consider this code:

```
function run()

   a=b*(sqrt(c)+ // 5 )

end function
```

After preprocessing, the preceding code contains errors and appears as follows:

```
function run()

   a=b*(sqrt(c)+

end function
```

8.1.2 Identifiers

An identifier is a sequence of characters used to denote one of the following:

- Variable name
- Object name
- Function name
- Label name

Syntax

identifier : *{leter}({leter}|{digit})**

leter : one of

_ a b c d e f g h i j k l m

n o p q r s t u v w x y z

A B C D E F G H I J K L M

N O P Q R S T U V W X Y Z

digit : one of

0 1 2 3 4 5 6 7 8 9

8.1.3 MAXL Keywords

Keywords are predefined reserved identifiers that have special meanings. They cannot be used as identifiers in your program. The following keywords are reserved for MAXL:

Syntax

keyword: one of

break	case	continue	else
elseif	end	exit	float
for	function	global	goto
if	initevent	int	otherwise
reset	return	switch	TRACE
wait	waitevent	while	

@perror @emcy

8.1.4 Punctuators

Punctuators in MAXL have syntactic and semantic meaning to the compiler but do not, of themselves, specify an operation that yields a value.

Syntax

punctuator : one of

!	%	^	&	*	(	)	-	+	=	/
|	~	[	]	;	:	"	<	>	,	#

The punctuators **[ ]** and **( )** must appear in pairs.

8.1.5 Operators

Operators specify an evaluation to be performed on one of the following:

- One operand (unary operator)
- Two operands (binary operator)

Table 1.1 lists the operators available in MAXL.

Operators follow a strict precedence, which defines the evaluation order of expressions containing these operators. Operators associate with either the expression on their left or the expression on their right; this is called "associativity." Operators in the same group have equal precedence and are evaluated left to right in an expression unless explicitly forced by a pair of parentheses, (). Table 1.1 shows the precedence and associativity of MAXL operators (from highest to lowest precedence).

Table 1.1 Operator Precedence and Associativity

Operator	Name or Meaning	Associativity
( )	Function call	Left to right
.	Member selection (object)	Left to right
+	Unary plus	None
-	Arithmetic negation (unary)	None
~	Bitwise complement	None
*	Multiplication	Left to right
/	Division	Left to right
%	Remainder (modulus)	Left to right
+	Addition	Left to right
-	Subtraction	Left to right
<<	Left shift	Left to right
>>	Right shift	Left to right
<	Less than	Left to right
>	Greater than	Left to right
<=	Less than or equal to	Left to right
>=	Greater than or equal to	Left to right

==	Equality	Left to right
!=	Inequality	Left to right
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
|	Bitwise OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
=	Assignment	Right to left
,	Comma	Left to right

8.1.6 Literals

Invariant program elements are called "literals" or "constants." The terms "literal" and "constant" are used interchangeably here. Literals fall into four major categories: integer, floating-point, and string literals.

Syntax

literal :

integer-constant

floating-constant

string-literal

8.1.6.1 Integer Constant

Integer constants are constant data elements that have no fractional parts or exponents. They always begin with a digit. You can specify integer constants in decimal, octal, or hexadecimal form. They can specify signed or unsigned types and long or short types.

Syntax

integer-constant :

decimal-constant integer

hexadecimal-constant integer

decimal-constant :

decimal-constant digit

hexadecimal-constant :

0x *hexadecimal-digit*

0X *hexadecimal-digit*

decimal-constant digit: one of

0 1 2 3 4 5 6 7 8 9

hexadecimal-digit : one of

0 1 2 3 4 5 6 7 8 9

a b c d e f

A B C D E F

To specify integer constants using octal or hexadecimal notation, use a prefix that denotes the base. To specify an integer constant of a given integral type, use a suffix that denotes the type.

To specify a decimal constant, begin the specification with a nonzero digit. For example:

```
i = 157;   // Decimal constant
```

To specify a hexadecimal constant, begin the specification with 0x or 0X (the case of the "x" does not matter), followed by a sequence of digits in the range 0 through 9 and a (or A) through f (or F). Hexadecimal digits a (or A) through f (or F) represent values in the range 10 through 15. For example:

```
i = 0x3fff;   // Hexadecimal constant

j = 0X3FFF;   // Equal to i
```

8.1.6.2 Floating-Point Constant

Floating-point constants specify values that must have a fractional part. These values contain decimal points (.) and can contain exponents.

Syntax

floating-constant :

fractional-constant exponent-part$_{opt}$
digit-sequence exponent-part

fractional-constant :

digit-sequence$_{opt}$. *digit-sequence*
digit-sequence .

exponent-part :

e *sign*$_{opt}$ *digit-sequence*
E *sign*$_{opt}$ *digit-sequence*

sign : one of

+ –

digit-sequence :

digit
digit-sequence digit

Floating-point constants have a "mantissa," which specifies the value of the number, an "exponent," which specifies the magnitude of the number, and an optional suffix that specifies the constant's type. The mantissa is specified as a sequence of digits followed by a period, followed by an optional sequence of digits representing the fractional part of the number. For example:

```
18.46

38.
```

The exponent, if present, specifies the magnitude of the number as a power of 10, as shown in the following example:

```
18.46e0     // 18.46

18.46e1     // 184.6
```

If an exponent is present, the trailing decimal point is unnecessary in whole numbers such as 18E0.

8.1.6.3 String Literals

A string literal consists of one or more characters from the source character set surrounded by double quotation marks ("). A string literal represents a sequence of characters that, taken together, form a null-terminated string.

Syntax

string-literal :

"*s-char-sequence*"

s-char-sequence :

s-char

s-char-sequence s-char

s-char :

any member of the source character set except the double quotation mark (")

8.2 Basic Concepts

8.2.1 Declarations and Definitions

Declarations tell the compiler that a program element or name exists. Definitions specify what code or data the name describes.

8.2.1.1 Declarations

A declaration introduces one or more names into a program. Now MAXL supports following syntactic of declarations:

- Variable declaration
- Function declaration
- Global variable declaration inside function body

Function declarations also serve as definitions.

Global variable declaration inside function body specifies by **global** keyword.

Examples:

```
int a,b // declaration global variables

function run() // begin of function declaration

        global a // global variable declaration inside function body

        int c  // local variable declaration in function man()

        c=56

        a=b+c // !!!error!!! variable b not declared in this function

        a=c+17

end function // end of function declaration
```

8.2.1.2 Definitions

A definition is a unique specification of a function or callback. Because definitions must be unique, a program can contain only one definition for a given program element.

8.2.2 Program

A program consists of one-translation unit. Execution (conceptually) begins in the translation unit that contains the function run.

MAXL program consist of following parts:

- Global variables declarations
- Functions and callback definitions
    - Declaration global variables for access inside function
    - Declaration local variables
    - Program lines.

8.2.3 Startup and Termination

Program startup is facilitated by function: run. Termination program is facilitated by end of **run** function or exit keyword.

8.2.3.1 Program Startup – the run Function

A special function called **run** is the entry point to all MAXL programs. The compiler does not predefine this function; rather, it must be supplied in the program text. The declaration syntax for **run** is:

```
function run( )

//… Some code
```

```
end function
```

8.2.3.2 Program Termination

In MAXL, there are several ways to exit a program:

- Call the **exit** operator.
- Finish program by finish **run** function.
- Call the **return** operator from function **run**.

8.2.3.2.1 exit Operator

Operator **exit** immediately finished program flow.

8.2.3.2.2 return Operator

Issue a **return** operator from **run** is functionally equivalent to using the **exit** operator. Consider the following example:

```
function run()
    int i
    …some code…
    if(i>2)
        exit
        return
    end if
    …some code…
end function
```

The **exit** and **return** statements in the preceding example are functionally identical.

8.2.4 Types

MAXL supports three kinds of object types:

- Fundamental types are built into the language (such as **int** and **float**). Instances of these fundamental types are called "variables."
- Object types are build-in types with encapsulated methods and properties (such as **axis** and **vector**).
- Debug string used for formatting debug messages

8.2.4.1 Fundamental Types

Fundamental types in MAXL are divided into two categories: integral and floating.

The following table explains the restrictions on type sizes.

Fundamental Types of the MAXL

Category	Type	Contents
Integral	**int**	32 bits integer type.
Floating	**float**	64 bits float point type. Like *double* in C/C++

8.2.4.2 Object types

Object is a base unit of the Maestro.

There are some different types of MAXL objects:

- Axis
- Vector
- Group
- Trajectory
- Program

Each of these objects defined by its name and uses specific set of attributes. This attributes set depend of object type.

MAXL supports two kinds of object attributes:

- Command
    - Getter
    - Setter
    - Getter/Setter
    - Executer
- Method

8.3 Standard Conversions

- Auto casting from integer to float and vice versa is performed.

For example:

```
int i

float f

i = 5.5  // after this program line executing variable i contain 5

f = i    // after this program line executing variable f contain 5 too
```

- Type of any binary operation defining according to operands type:

First operand type	Second operand type	Result type
Integer	Integer	Integer
Float	Integer	Float
Integer	Float	Float
Float	Float	Float

- Type of any logical operation not depends of operands type and always integer.

  If result of logical operation is true, result value is 1 else 0.

- Operator '%'(modulo) used only for integer operands.
- Bitwise operators use only for integer operands.

8.4 Expressions

Expressions are sequences of operators and operands that are used for one or more of these purposes:

- Computing a value from the operands.
- Designating objects or functions.
- Generating "side effects." (Side effects are any actions other than the evaluation of the expression — for example, modifying the value of an object.)

8.4.1 Types of Expressions

MAXL expressions are divided into several categories:

- Primary expressions. These are the building blocks from which all other expressions are formed.
- Postfix expressions. These are primary expressions followed by an operator — for example, the array subscript or postfix increment operator.
- Expressions formed with unary operators. Unary operators act on only one operand in an expression.
- Expressions formed with binary operators. Binary operators act on two operands in an expression.
- Logical Operators. Used to combine multiple conditions formed using relational or equality expressions.
- Simple Assignment.

8.4.1.1 MAXL Primary Expressions

Primary expressions are the building blocks of more complex expressions. They are literals and names.

Syntax

primary-expression :

literal

(*expression*)

name

A *literal* is a constant primary expression. Its type depends on the form of its specification. See Literals for complete information about specifying literals.

An expression enclosed in parentheses is a primary expression whose type and value are identical to those of the unparenthesized expression.

8.4.1.1.1 Names

In the MAXL syntax for *primary-expression*, a *name* is a primary expression that can appear only after the member-selection operators and names the attribute of an object.

Syntax

name :

identifier

object . *attribute*

8.4.1.2 Postfix Expressions

Postfix expressions consist of primary expressions or expressions in which postfix operators follow a primary expression.

The postfix operators are listed in Table:

Table Postfix Operators

Operator Name	Operator Notation
Function-call operator	**( )**
Method-selection operator	**.**
Postfix increment operator	**++**
Postfix decrement operator	**– –**

Syntax

postfix-expression :

function-name **(** *expression-list*$_{opt}$ **)**

object **.** *method* **(***expression-list*$_{opt}$ **)**

name **++**

name **– –**

expression-list :

expression

expression-list **,** *expression*

8.4.1.2.1 Function-Call Operator

A *postfix-expression* followed by the function-call operator, **( )**, specifies a function call. The arguments to the function-call operator are zero or more expressions separated by commas — the actual arguments to the function.

8.4.1.3 Expressions with Unary Operators

Unary operators act on only one operand in an expression. The unary operators are:

- Unary plus operator (+)
- Unary negation operator (–)
- Logical NOT operator (!)
- One's complement operator (~)

These operators have right-to-left associativity.

Syntax

unary-expression :

postfix-expression

++ name

– – name

unary-operator expression

unary-operator : one of

+ – ! ~

8.4.1.3.1 Unary Plus Operator (+)

The result of the unary plus operator (**+**) is the value of its operand. The operand to the unary plus operator must be of an arithmetic type.

Integral promotion is performed on integral operands. The resultant type is the type to which the operand is promoted

8.4.1.3.2 Unary Negation Operator (-)

The unary negation operator (**–**) produces the negative of its operand. The operand to the unary negation operator must be an arithmetic type.

8.4.1.3.3 Logical NOT Operator (!)

The result of the logical NOT operator (**!**) is 0 if its operand evaluates to a nonzero value; the result is 1 only if the operand is equal to 0. The operand must be of arithmetic or pointer type. The result is of type **int**.

For an expression *e*, the unary expression **!***e* is equivalent to the expression **(***e* **==** 0).

The following example illustrates the logical NOT operator (**!**):

```
if( !(x < y) )
```

If x is greater than or equal to y, the result of the expression is 1 (true). If x is less than y, the result is 0 (false).

Unary arithmetic operations on pointers are illegal.

8.4.1.3.4 One's Complement Operator (~)

The one's complement operator (~), sometimes called the "bitwise complement" operator, yields a bitwise one's complement of its operand. That is, every bit that is set in the operand is 0 in the result. Conversely, every bit that is 0 in the operand is set in the result. The operand to the one's complement operator must be an integral type.

```
int y
y= 0xAAAA
 y = ~y
```

In this example, the new value assigned to y is the one's complement of the unsigned value 0xAAAA, or 0x5555.

Integral promotion is performed on integral operands, and the resultant type is the type to which the operand is promoted

8.4.1.4 Expressions with Binary Operators

Binary operators act on two operands in an expression. The binary operators are:

- Multiplicative operators
  - Multiplication (*)
  - Division (/)
  - Modulus (%)
- Additive operators
  - Addition (+)
  - Subtraction (–)
- Shift operators
  - Right shift (>>)
  - Left shift (<<)
- Relational and equality operators
  - Less than (<)

- Greater than (>)
- Less than or equal to (<=)
- Greater than or equal to (>=)
- Equal to (==)
- Not equal to (!=)

- Bitwise operators
  - Bitwise AND (&)
  - Bitwise exclusive OR (^)
  - Bitwise inclusive OR (|)
  - Logical AND (&&)
  - Logical OR (||)

8.4.1.4.1 MAXL Multiplicative Operators

The multiplicative operators are:

- Multiplication (*)
- Division (/)
- Modulus or "remainder from division" (%)

These binary operators have left-to-right associativity.

Syntax

multiplicative-expression :

expression * *expression*

expression / *expression*

expression % *expression*

expression:

multiplicative-expression

The multiplicative operators take operands of arithmetic types. The modulus operator (%) has a stricter requirement in that its operands must be of integral type.

The multiplication operator yields the result of multiplying the first operand by the second.

The division operator yields the result of dividing the first operand by the second.

The modulus operator yields the remainder given by the following expression, where *e1* is the first operand and *e2* is the second: *e1* – (*e1* / *e2*) * *e2*, where both operands are of integral types.

Division by 0 in either a division or a modulus expression is undefined and causes a run-time error. Therefore, the following expressions generate undefined, erroneous results:

```
i % 0
f / 0.0
```

If both operands to a multiplication, division, or modulus expression have the same sign, the result is positive. Otherwise, the result is negative. The result of a modulus operation's sign is implementation-defined

8.4.1.4.2 MAXL Additive Operators

The additive operators are:

- Addition (+)
- Subtraction (–)

These binary operators have left-to-right associativity.

Syntax

additive-expression :

expression + *expression*

expression – *expression*

expression:

additive-expression

The additive operators take operands of arithmetic type. The result of the addition (+) operator is the sum of the operands. The result of the subtraction (–) operator is the difference between the operands.

8.4.1.4.3 MAXL Shift Operators

The bitwise shift operators are:

- Right shift (>>)
- Left shift (<<)

These binary operators have left-to-right associativity.

Syntax

shift-expression :

expression << *expression*

expression >> *expression*

expression:

additive-expression

Both operands of the shift operators must be of integral types. The type of the result is the same as the type of the left operand. The value of a right-shift expression *e1* >> *e2* is *e1* / 2^{e2}, and the value of a left-shift expression *e1* << *e2* is *e1* * 2^{e2}.

The results are undefined if the right operand of a shift expression is negative or if the right operand is greater than or equal to the number of bits in the (promoted) left operand.

The left-shift operator causes the bit pattern in the first operand to be shifted left the number of bits specified by the second operand. Bits vacated by the shift operation are zero-filled. This is a logical shift, as opposed to a shift-and-rotate operation.

The right-shift operator causes the bit pattern in the first operand to be shifted right the number of bits specified by the second operand. Bits vacated by the shift operation are zero-filled for unsigned quantities. For signed quantities, the sign bit is propagated into the vacated bit positions. The shift is a logical shift if the left operand is an unsigned quantity; otherwise, it is an arithmetic shift

8.4.1.4.4 MAXL Relational and Equality Operators

The relational and equality operators determine equality, inequality, or relative values of their operands. The relational operators are shown in Table

Table Relational and Equality Operators

Operator	Meaning
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to

>=	Greater than or equal to

8.4.1.4.4.1 Relational Operators

The binary relational operators determine the following relationships:

- Less than
- Greater than
- Less than or equal to
- Greater than or equal to

Syntax

relational-expression :

expression < *expression*

expression > *expression*

expression <= *expression*

expression >= *expression*

expression:

relational-expression

The relational operators have left-to-right associativity. Both operands of relational operators must be of arithmetic type. They yield values of type int. The value returned is 0 if the relationship in the expression is false; otherwise, it is 1.

8.4.1.4.4.2 Equality Operators

The binary equality operators compare their operands for strict equality or inequality.

Syntax

equality-expression :

expression == *expression*

expression != *expression*

expression:

equality-expression

The equality operators, equal to (==) and not equal to (!=), have lower precedence than the relational operators, but they behave similarly.

The equal-to operator (==) returns 1 if both operands have the same value; otherwise, it returns 0. The not-equal-to operator (!=) returns 1 if the operands do not have the same value; otherwise, it returns 0.

Equality operators can compare pointers to members of the same type.

8.4.1.4.5 MAXL Bitwise Operators

The bitwise operators are:

- Bitwise AND (&)
- Bitwise exclusive OR (^)
- Bitwise inclusive OR (|)

These operators return bitwise combinations of their operands

8.4.1.4.5.1 Bitwise AND Operator

The bitwise AND operator (&) returns the bitwise AND of the two operands. All bits that are on (1) in both the left and right operand are on in the result; bits that are off (0) in either the left or the right operand are off in the result.

Syntax

and-expression :

expression & expression

expression:

and-expression

Both operands to the bitwise AND operator must be of integral types

8.4.1.4.5.2 Bitwise Exclusive OR Operator

The bitwise exclusive OR operator (^) returns the bitwise exclusive OR of the two operands. All bits that are on (1) in either the left or right operand, but not both, are on in the result. Bits that are the same (either on or off) in both operands are off in the result.

Syntax

exclusive-or-expression :

expression ^ expression

expression:

exclusive-or-expression

Both operands to the bitwise exclusive OR operator must be of integral types.

8.4.1.4.5.3 Bitwise Inclusive OR Operator

The bitwise inclusive OR operator (|) returns the bitwise inclusive OR of the two operands. All bits that are on (1) in either the left or right operand is on in the result. Bits that are off (0) in both operands are off in the result.

Syntax

inclusive-or-expression :

expression | *expression*

expression:

inclusive-or-expression

Both operands to the bitwise inclusive OR operator must be of integral types.

8.4.1.5 MAXL Logical Operators

The logical operators, logical AND (&&) and logical OR (||), are used to combine multiple conditions formed using relational or equality expressions.

8.4.1.5.1 Logical AND Operator

The logical AND operator (&&) returns the integral value 1 if both operands are nonzero; otherwise, it returns 0. Logical AND has left-to-right associativity.

Syntax

logical-and-expression :

expression && expression

expression:

logical-and-expression

The operands to the logical AND operator must be of integral type. The operands are commonly relational or equality expressions.

The first operand is completely evaluated and all side effects are completed before continuing evaluation of the logical AND expression.

8.4.1.5.2 Logical OR Operator

The logical OR operator (||) returns the integral value 1 if either operand is nonzero; otherwise, it returns 0. Logical OR has left-to-right associativity.

Syntax

logical-or-expression :

expression || expression

expression:

logical-or-expression

The operands to the logical OR operator must be of integral type. The operands are commonly relational or equality expressions.

The first operand is completely evaluated and all side effects are completed before continuing evaluation of the logical OR expression.

8.4.1.6 Simple Assignment

The simple assignment operator (=) causes the value of the second operand to be stored in the object specified by the first operand. If both objects are of arithmetic types, the right operand is converted to the type of the left, prior to storing the value.

8.4.2 Semantics of Expressions

This section explains when, and in what order, expressions are evaluated.

8.4.2.1 Order of Evaluation

This section discusses the order in which expressions are evaluated but does not explain the syntax or the semantics of the operators in these expressions.

Consider this example:

```
function  run()
   int a , b , c, res1, res2, res3
   a = 2
   b = 4
   c = 9
   res1 = a + b * c
   res2 = a + (b * c)
   res3 = (a + b) * c
end function
```

The output from the preceding code is:

```
res1 = 38
res2 = 38
res3 = 54
```

The order in which the res1 is evaluated is determined by the precedence and associativity of the operators:

1. Multiplication (*) has the highest precedence in this expression; hence the subexpression `b * c` is evaluated first.

2. Addition (+) has the next highest precedence, so `a` is added to the product of `b` and `c`.

3. Assignment operator (=) has the lowest precedence in the expression.

When parentheses are used to group the subexpressions, they alter the precedence and also the order in which the expression is evaluated

8.5 Statements

MAXL statements are the program elements that control how and in what order objects are manipulated.

MAXL statements are executed sequentially, except when an expression statement, a selection statement, an iteration statement, or a jump statement specifically modifies that sequence.

Syntax

statement:

declaration-statement
labeled-statement
expression-statement
selection-statement
iteration-statement
jump-statement

8.5.1 Labeled Statements

To transfer program control directly to a given statement, the statement must be labeled.

Syntax

labeled-statement:

##identifier statements

case (*expression) statements*

otherwise *statements*

8.5.1.1 Using Labels with the goto Statement

The appearance of an *identifier* label in the source program declares a label. Only a **goto** statement can transfer control to an *identifier* label. The following code fragment illustrates use of the **goto** statement and an *identifier* label to escape a nested loop:

```
for ( i = 0 : 2 : 20 )
    for( j = 2 : 16  )
      if( i > j+10)
          goto ##lab
      end if
    end for
end for

## label
   I=0
```

A label cannot appear by itself but must always be attached to a statement.

The label has function scope and cannot be redeclared within the function. However, the same name can be used as a label in different functions.

8.5.1.2 Using Labels in the case Statement

Labels that appear after the **case** keyword cannot also appear outside a **switch** statement. (This restriction also applies to the **otherwise** keyword.) The following code fragment shows the correct use of **case** labels:

```
switch( cmd )
case(1) //begin motion
            axis.bg
case(2) // motor On
            axis.mo = 1
case(3) // motor Off
            axis.mo = 0
otherwise  // get motor status
            cmd = axis.ms
```

```
end switch
```

8.5.2 Selection Statements

The MAXL selection statements, if and switch, provide a means to conditionally execute sections of code.

Syntax

selection-statement:

if (*expression*) *statement* **end [if]**

if (*expression*) *statement* **else** *statement* **end [if]**

switch (*expression*) *statement* **end [switch]**

8.5.2.1 The MAXL if Statement

The **if** statement evaluates the expression enclosed in parentheses. The expression must be of arithmetic type.

In both forms of the **if** syntax, if the expression evaluates to a nonzero value (true), the statement dependent on the evaluation is executed; otherwise, it is skipped.

In the **if...else** syntax, the second statement is executed if the result of evaluating the expression is zero.

```
if (b == 4 )
   if( c == 18 )
      a=1 // b is 4; c is 18
   else
      a=2 // b is 4; c isn`t 18
   end if
else
   a=3 // b isn`t 4;
end if
```

8.5.2.2 The MAXL switch Statement

The MAXL **switch** statement allows selection among multiple sections of code, depending on the value of an expression.

The **switch** statement causes an unconditional jump to, into, or past the statement that is the "switch body," depending on the value of the controlling expression, the values of the **case** labels, and the presence or absence of a **otherwise** label.

The **switch** body is normally a compound statements (although this is not a syntactic requirement). Usually, some of the statements in the switch body are labeled with **case** labels or with the **otherwise** label. Labeled statements are not syntactic requirements, but the switch statement is meaningless without them. The **otherwise** label can appear only once.

Syntax

case *(expression) statements*

otherwise *statements*

The expression in the **case** label is converted to the type of the controlling expression and is then compared for equality. The behavior is shown in Table.

Table Switch Statement Behavior

Condition	Action
Converted value matches that of the promoted controlling expression.	Control is transferred to the statement following that label.
None of the constants match the constants in the **case** labels; **otherwise** label is present.	Control is transferred to the **otherwise** label.
None of the constants match the constants in the **case** labels; **otherwise** label is not present.	Control is transferred to the statement after the **switch** statement.

A switch statement can be nested. In such cases, **case** or **otherwise** labels associate with the most deeply nested **switch** statements that enclose them. For example:

```
switch( attribute_type)
case (1)      // Object attribute is command.
   switch( cmdName)
      case (1)   // Begin motion command.
        Axis.bg
      case( 2)  // Motor On command.
```

```
        Axis.mo=1
   end switch
case (2)      // Object attribute is method.
   ...
otherwise
   TRACE (“  Error attribute ”  )
end switch
```

The preceding code fragment shows how switch statements can be nested.
The **switch** statement that selects on the value of cmdName is executed only if attribute is 1(is command). The **case** labels for menu selections, 1 (BG) and 2 (MO=1), associate with the inner **switch** statement.

8.5.3 Iteration Statements

Iteration statements cause statements (or compound statements) to be executed zero or more times, subject to some loop-termination criteria. When these statements are compound statements, they are executed in order, except when either the **break** statement or the **continue** statement is encountered. For a description of these statements, see The break Statement and The continue Statement.

MAXL provides two iteration statements — **while** and **for**. Each of these iterates until its termination expression evaluates to zero (false), or until loop termination is forced with a **break** statement.

Syntax

iteration-statement :

while (*expression*) *statements* **end while**

for *identifier-variable*=*expression* : *[step- expression* :*] terminat-expression*

statements

end for

8.5.3.1 The MAXL while Statement

The **while** statement executes a statement repeatedly until the termination condition (the expression) specified evaluates to zero.

The test of the termination condition takes place before each execution of the loop; therefore, a while loop executes zero or more times, depending on the value of the termination expression. For example:

```
while( attribute < 56)
        attrbute=attribute+17
        TRACE(“  Attribute : %”  , attribute)
end while
```

8.5.3.2 The MAXL for Statement

The **for** statement can be divided into three separate parts, as shown in

Table for Loop Elements

Syntax Name	When Executed	Contents
identifier-variable = *expression*	Before any other element of the **for** statement or the substatement.	Used to initialize loop indices.
step-expression	Before execution of a given iteration of the loop, excluding the first iteration.	Calculation iteration step. If this field is absent iteration step is 1.
terminate-expression	At the end of each iteration of the loop.	Used as loop-termination criteria.

The **for**-statement executes the *statements* repeatedly until *identifier-value* less or equal to *terminate-expression* value. The *step-expression* field is optional. If *step-expression* has a negative value *for-statement* executes the statements repeatedly until *identifier-value* great or equal to *terminate-expression* value. For example:

```
for i=1:1:10
        test(i)
end for
```

8.5.4 Jump Statements

The MAXL jump statements perform an immediate local transfer of control.

Syntax

jump-statement:

break

continue

return

goto identifier

8.5.4.1 The MAXL break Statement

The **break** statement is used to exit an iteration statement. It transfers control to the statement immediately following the iteration statement.

The **break** statement terminates only the most tightly enclosing loop. In loops, **break** is used to terminate before the termination criteria evaluate to 0. The following example illustrates the use of the **break** statement in a **while** loop:

```
While(1)    // No termination condition.
  if( program.status() == 2) //program halt
      break
end while
```

Note There are other simple ways to escape a loop. It is best to use the **break** statement in more complex loops, where it can be difficult to tell whether the loop should be terminated before several statements have been executed.

8.5.4.2 The MAXL continue Statement

The **continue** statement forces immediate transfer of control to the loop-continuation statement of the smallest enclosing loop. (The "loop-continuation" is the statement that contains the controlling expression for the loop.) Therefore, the **continue** statement can appear only in the dependent *statement* of an iteration statement.

8.5.4.3 The MAXL return Statement

The **return** statement allows a function to immediately transfer control back to the calling function.

A function can have any number of **return** statements.

8.5.4.4 The goto Statement

The **goto** statement performs an unconditional transfer of control to the named label. The label must be in the current function.

8.5.5 Declaration Statements

Declaration statements introduce new names into the current function. Before using global variable inside function it must be declared by keyword **global**.

Syntax

declaration-statement:

global type name-list

type name-list

type:

int

float

name-list:

name

name-list , name

Multiple declarations of the same name in the same function are illegal.

8.6 Functions

8.6.1 Function definition

A function definition specifies the name of the function, the types and number of arguments it expects to receive, and types and number of return arguments. A function definition also includes a function body with the declarations of its local variables, and the statements that determine what the function does.

function-definition:

function function-name (argument-list)

function-body

end function

function type variable-name = function-name (argument-list)

function-body

end function

function [argument-list] = function-name (argument-list)

function-body

end function

argument-list:

type name

argument-list , type name

function-body:

declaration-statements statements

Examples:

```
function int max_value = MAX_INT(int a, int b, int c)
        int max_a_b
        if(a>b)
                max_a_b = a
        else
                max_a_b = b
        end if
        if(max_a_b > c)
                max_value = max_a_b
        else
                max_value = c
        end if
end function
```

This is function with name – MAX_INT has three integer input arguments (a, b, c) and one integer return argument (max_value). Function body includes one *declaration-statement* (local integer variable *max_a_b*) and two statements *if-else*.

```
function [int min1, int min2] = MIN_2_INT(int a, int b, int c)
        int max_a_b
        int min_a_b
        if(a>b)
                max_a_b = a
                min_a_b = b
        else
                max_a_b = b
                min_a_b = a
        end if
        if(max_a_b > c)
                min1 = min_a_b
                min2 = c
        else
                min1 = a
                min2 = b
        end if
 end function
```

This is function with name – MIN_2_INT has three integer input arguments (a, b, c) and two integer return arguments (min1, min2). Function body includes two *declaration-statements* (local integer variables *max_a_b* and *min_a_b*) end two statements *if-else*.

If function hasn't return arguments it's header not contain type and name return argument. For example:

```
function foo()
// to do something' s
……
end function
```

This function hasn't input and returns arguments.

8.6.2 Build-in functions

MAXL has predefinition functions.

Function name	Input argument(s)	Return argument(s)	Remarks
sin	float	float	
cos	float	float	

tan	float	float	Trigonometry functions, input argument angle value in radians.
asin	float	float	
acos	float	float	
atan	float	float	
exp	float	float	Calculates the exponential function value of a floating-point number.
pow	float x, float y	float	Calculates the value of x raised to the power of y, x^y.
log	float	float	Calculates the natural logarithm (base e) of a floating-point number
log10	float	float	Calculates the common logarithm (base 10) of a floating-point number.
min	float a, float b	float	Returns minimal value
max	float a, float b	float	Returns maximal value
abs	float	float	Calculates the absolute value of a floating-point number.
sqrt	float	float	Calculates the square root of a floating-point number.
Round	float	int	Rounds float value to integer.
Floor	float	float	Calculates the floor (greatest integer less than or equal to) value of a number.
Ceil	float	float	Calculates the ceiling (smallest integer greater than or equal to) value of a number. This is the result of rounding up.
Sync	int busId, float delay	-	Starts CanOPEN SYNC mechanism. BusID – bus identifier [1,2]. Delay in milliseconds, minimal value 1.
Tstamp	int busId, int sync_count		Defines TIME-STAMP interval in number SYNC messages. Default value 50.

8.6.3 Callback (interrupt) functions

MAXL supports callback function mechanism. Callback – this is function that calls by system, not by program. Name of MAXL callback functions are begin from symbol '@'.

MAXL supports the following list of callback functions:

- @perror(int error_number) – calls then a runtime program error occur, *error_number* contains error code.

  When runtime error occurs the MAXL program status will be set to ABORT. In case of declaration *@perror* callback function this function will be called first and program finished with HALT status. Use **restart** keyword in *@perror* callback function to restart program.

- @emcy(int busID, int nodeID, int first4bytes, int second4bytes) – call than CanOpen EMERGENCY message appears on the bus.
  - *BusID* – CAN bus identifier [0,1].
  - *NodeID* – CAN node identifier [1,127].
  - *First4bytes* – first 4 bytes CAN message data.
  - *Second4bytes* – last 4 bytes CAN message data.
- @emcy(int nodeID, int first4bytes, int second4bytes)
  - *NodeID* – CAN node identifier [1,127].
  - *First4bytes* – first 4 bytes CAN message data.
  - *Second4bytes* – last 4 bytes CAN message data.
- @emcy(int nodeID, int first4bytes)
  - *NodeID* – CAN node identifier [1,127].
  - *First4bytes* – first 4 bytes CAN message data.

8.7 Virtual Machine Control Statements

8.7.1 *wait* control statement

The *wait* keyword suspends the execution of the program until the specified time is elapsed.

The expression may be within round parentheses or without ones.

The expression specifies the waiting time in milliseconds. It can be numerical expression only, which is evaluated in a single value.

Example:

```
function foo()
// to do something' s
……
wait(1000) // program waiting one second
```

```
end function
```

8.7.2 *TRACE* control statement

For throw out message from virtual machine to Elmo Studio uses program statement with *TRACE* keyword. Elmo Studio – this is standard IDE for development and debugging MAXL programs. If Maestro's virtual machine comes across statement with *TRACE* during debugging MAXL programs, in output window of Elmo Studio appears message formatted in its statement.

8.7.3 *reset* control statement

The *reset* keyword uses for restart program. After program statement with operator *reset* program flow will be restarted from *run* function. At the same time all call function stack resets.

Chapter 9: The Maestro API

Use MAC_GetLastError to any function that has a return value != 0 (see example) to retrieve error string.

Arbitrations

MAS/MAC – Multi Axes Supervisor \ Multi Axes Controller synonyms for the Maestro

Interfaces:

API – (Application Program Interface) Set of functions that enables a user's program to have communication and other requests with the Maestro and the connected subset drives.

PVT – Position Velocity Time, motion method commonly used by the Maestro to perform a movement.

Multi-axes supervisor API Library Operation System Requirements:
Windows NT/2000/XP: Requires Windows NT 4.0 or later;
Windows 95/98/Me: Requires Windows 98, Me (or Windows 95 with DCOM).

MAC_Initialize

This function enables the user to initialize Multi-axes supervisor API Library

Syntax:

`int MAC_Initialize`(DWORD dwComInitialize, BOOL bEventInitialize);

Input Parameters:

DWORD dwComInitialize	Initialize COM Mechanism Parameter determines the concurrency model used for incoming calls to objects created by this thread. This parameter can be one of the following: COM_INIT_DISABLE - disable inner COM initialization, COM_INIT_SINGLETHREADED - single-threaded model, COM_INIT_MULTITHREADED - multi-threaded model
BOOL bEventInitialize	Initialize MAS Event Mechanism

Return Value:

int - initialization status:

0	OK;
1	The API library is already initialized;
2	The COM library is already initialized on this thread;
3	A previous COM initialization specified a different apartment model for this thread;

4 The COM library initialization failed;

5 The MAS Events mechanism initialization failed;

Use MAC_GetLastError to any function that has a return value != 0 (see example) to retrieve the error string.

Remarks:

MAC_Initialize() can initializes the COM library for use by the calling thread, sets the thread's concurrency model, and creates a new apartment for the thread if one is required. There are two types of apartments: single-threaded apartments, and multithreaded apartments.

- Single-threaded Apartments—Single-threaded apartments consist of exactly one thread, so all COM objects that live in a single-threaded apartment can receive method calls only from the one thread that belongs to that apartment. All method calls to a COM object in a single-threaded apartment are synchronized with the windows message queue for the single-threaded apartment's thread. A process with a single thread of execution is simply a special case of this model.
- Multithreaded Apartments—Multithreaded apartments consist of one or more threads, so all COM objects that live in a multithreaded apartment can receive method calls directly from any of the threads that belong to the multithreaded apartment. Threads in a multithreaded apartment use a model called *free-threading*. Calls to COM objects in a multithreaded apartment are synchronized by the objects themselves.

Sample:

```
int nOk = MAC_Initialize( COM_INIT_SINGLETHREADED, FALSE );
```

MAC_Uninitialize

This function enables the user to uninitialize Multi Axes Supervisor API Library.

Syntax:

void MAC_Uninitialize();

Sample:

```
MAC_Uninitialize();
```

MAC_CreateTCPConnection

This function enables the user to initiate TCP/IP communication with the Multi-axes supervisor.

Syntax:

```
COMMUNICATION_HANDLE MAC_CreateTCPConnection (LPCTSTR cszIP,
                                              DWORD
                                              dwTimeout/*=5000*/,
                                              WORD wPort/*=23*/ );
```

Input Parameters:

LPCTSTR cszIP	IP Address of supervisor
DWORD dwTimeout	Max time for communications to be established
WORD wPort	Communication port

Return Value:

COMMUNICATION_HANDLE	Multi-axes supervisor communication handle, if connection failed - NULL.

Sample:

```
COMMUNICATION_HANDLE hCon = MAC_CreateTCPConnection
                            ( _T("10.10.10.112"), 5000, 23 );
```

MAC_CreateRS232Connection

Note:

This interface is not implanted yet.

This function enables the user to initiate RS232 communication with the Multi-axes supervisor.

Syntax:

```
COMMUNICATION_HANDLE MAC_CreateRS232Connection( BYTE btPort,
                                                UINT uBautRate  /*=19200*/,
                                                BYTE btParity   /*=0*/,
                                                BYTE btDataBits /*=8*/,
                                                BYTE btStopBits /*=1*/,
                                                BYTE btFlowCtrl /*=0 */);
```

Input Parameters:

BYTE btPort - COM port number, value: 1,2,3,4,5,6,7,8

UINT uBaudRate - Current baud rate:

4800
9600
19200
38400
57600
115200

BYTE btParity - Parity, one of the following:

0: No parity
1: Odd parity
2: Even parity
3: Mark
4: Space

BYTE btDataBits - Data bytes number, value: 4, 5, 6, 7, 8

BYTE btStopBits - Stop bytes number, value: 1, 1.5, 2

BYTE btFlowCtrl - Flow control, one of the following:

0: None
1: Xon/Xoff
2: Hardware

Return value:

COMMUNICATION_HANDLE Multi-axes supervisor communication handle, if connection failed - NULL.

Sample:

```
COMMUNICATION_HANDLE hCon = MAC_CreateRS232Connection( 1, 19200, 0,
                            8, 1, 0 );
```

MAC_CloseConnection

This function enables the user to close communication

Syntax:

`int MAC_CloseConnection`(COMMUNICATION_HANDLE hCommunication)

Input Parameters:

COMMUNICATION_HANDLE hCommunication

Return Value:

int Close connection status:

0	OK;
1	The API library is not initialized;
2	Zero Communication handle;
3	Bad Communication handle;

Use MAC_GetLastError to any function that has a return value != 0 (see example) to retrieve error string.

Sample:

```
if( !MAC_CloseConnection( hCon ) ) return FALSE;
```

MAC_SendCommand

This function enables the user to send commands to, and receive responses from, the connected Multi-axes supervisor.

Syntax:

```
Int MAC_SendCommand( COMMUNICATION_HANDLE hCommunication,
                     LPCTSTR      cszCmd,
                     LPTSTR       szOut,
                     WORD         wOutSize /*=1024*/,
                     DWORD        dwTimeout/*=1000*/);
```

Input Parameters:

`COMMUNICATION_HANDLE hCommunication`	Multi-axes supervisor communication handle.
`LPCTSTR cszCmd`	Input string, containing command or expression, to be sent to the MAS. This string may be a command to the SimplIQ drive connected to the Maestro or command to the Maestro itself.
`WORD wOutSize`	Size of string for response receiving.
`DWORD` dwTimeout	The maximum wait time for receiving a response from the drive, in milliseconds.

Output Parameter:

`LPTSTR szOut` String for response receiving.

Return Value:

`Int` Error Value:

- >0 OK
- >0 Communication error codes, user may use MAC_GetLastError to retrieve the error string.
  1. The API library is not initialized;
  2. Zero Communication Handle;
  3. Bad Communication Handle;
  4. Unable to create interface ITNElmoPublic
  5. Send command failed;
- < 0 the string for reception of the answer is too small, value is the necessary size of a string (with a minus).

Sample:

```
retVal = MAC_SendCommand( hCon, _T("axis1.px=0"), szOut, 256,
1000 );
```

MAC_LocateDevices

This function enables the user to find Multi-axes supervisors.

Syntax:

```
int MAC_LocateDevices();
```

Return Value:

int - number of supervisors,

```
if < 0  error code:
    -1   The API library is not initialized;
    -2   Unable to create interface CLSID_MaestroLocator;
    -3   The MAS Locator failed;
    -4   The MAS Locator: unable to get device IP address;
    -5   The MAS Locator: unable to get device name;
```

Use MAC_GetLastError to any function that has a return value != 0 (see example) to retrieve error string.

Sample:

```
int number = MAC_LocateDevices();
```

MAC_GetDevice

This function enables the user to get Multi-axes supervisor parameters

Syntax:

`int MAC_GetDevice`(UINT uNum, LPTSTR szIP, LPTSTR szName);

Input Parameters:

UINT uNum Device number in this search results

Output Parameter:

LPTSTR szIP - IP address of supervisor (string with length 16 bytes minimum).

LPTSTR szName - Name of supervisor (string with length 32 bytes minimum).

Return Value:

int - command status: if 0 - OK,

else - error code:

1 - The API library is not initialized;

2 - The device is not found;

Use MAC_GetLastError to any function that has a return value != 0 (see example) to retrieve error string.

Sample:

```
int retVal = MAC_GetDevice( 0, szIP, szName );
```

MAC_GetIpByName

This function enables the user to get Multi-axes supervisor IP address by name

Syntax:

`int MAC_GetIpByName`(LPCTSTR cszName, LPTSTR szIP);

Input Parameters:

LPCTSTR cszName - Name of supervisor (string with length 32 bytes minimum).

Output Parameter:

LPTSTR szIP - IP address of supervisor (string with length 16 bytes minimum).

Return Value:

int - command status: if 0 - OK,

else	error code:
1	The API library is not initialized;
2	The device is not found;

Use MAC_GetLastError to any function that has a return value != 0 (see example) to retrieve error string.

Sample:

```
int retVal = MAC_GetIpByName( _T("MAS_1"), szIP );
```

MAC_IsDevicePresent

This function enables the user to check Multi-axes controller IP address

Syntax:

`int MAC_IsDevicePresent` (LPCTSTR cszHost, DWORD* pdwIP);

Input Parameters:

LPCTSTR cszHost - Host name or IP Address of controller.

Output Parameter:

DWORD* pdwIP - IP address of controller (NULL, if is Host not found).

Return Value:

int - command status: if 0 - OK,

else	error code:
1	The API library is not initialized;
2	Invalid input parameter: zero pointer;
3	Unable to create interface CLSID_MaestroLocator
4	The MAS Locator failed
5	The device is not found

Use MAC_GetLastError to any function that has a return value != 0 (see example) to retrieve error string.

Sample:

```
int retVal = MAC_IsDevicePresent( _T("10.10.10.52"), &dwIP );
```

MAC_LocateObjects

This function enables the user to start motion objects search

Motion object can be one of the following:

- Axis: Either Elmo_Axis or Axis_402 .
- Group: Several axes that may be addressed simultaneously.
- Vector: Is a group of two axes that perform a synchronized motion.

Syntax:

```
int MAC_LocateObjects( COMMUNICATION_HANDLE hCommunication,
                       MOTION_OBJECT_TYPE   moType );
```

Input Parameters:

COMMUNICATION_HANDLE hCommunication
Multi-axes supervisor communication handle.

MOTION_OBJECT_TYPE moType
motion object type mask for current search NULL - all motion objects

Return Value:

int - number of objects,

if < 0 error code:

-1 The API library is not initialized;

-2 Zero Communication Handle;

-3 Bad Communication Handle;

-4 Unable to create interface ITNElmoPublic;

-5 Send command failed ;

-6 The Command 'list' used internally by the function has failed;

Use MAC_GetLastError to any function that has a return value != 0 (see example) to retrieve error string.

Sample:

```
MOTION_OBJECT_TYPE moMask = MOT_AXIS|MOT_VECTOR;
int                nObjs  = MAC_LocateObjects( hCom, dwMask );
```

MAC_GetObject

This function enables the user to get motion object parameters

Syntax:

`int MAC_GetObject`(UINT uNum, MOTION_OBJECT* pObj);

Input Parameters:

UINT uNum - Object number in this search results

Output Parameter:

MOTION_OBJECT* pObj - Pointer to motion object data structure

Return Value:

int - Command status: if 0 - OK,
else - error code:
1 - The API library is not initialized;
2 - Invalid input parameter;
3 - The device is not found;

Use MAC_GetLastError to any function that has a return value != 0 (see example) to retrieve error string.

Sample:

```
int retVal = MAC_GetObject( 0, &moData );
```

MAC_DownloadTrajectory

This function enables the user to download trajectory to Multi-axes supervisor

A trajectory is a text file that contains a sequence of PVT points, created by the user.

Syntax:

```
int MAC_DownloadTrajectory( COMMUNICATION_HANDLE
                            hCommunication, LPCTSTR cszLocalPath );
```

Input Parameters:

COMMUNICATION_HANDLE hCommunication	Multi-axes supervisor communication handle.
LPCTSTR cszLocalPath	Path to trajectory file on desktop computer.

Return Value:

0 OK,

> 0 communication error codes:

1 The API library is not initialized;

2 Zero Communication handle;

3 Bad Communication handle;

4 Unable to create interface ITNElmoPublic;

5 Download Trajectory failed;

Use MAC_GetLastError to any function that has a return value != 0 (see example) to retrieve error string.

Sample:

```
int retVal = MAC_DownloadTrajectory( hCom, szPath );
```

MAC_RemoveTrajectory

This function enables the user to remove trajectory from Multi-axes supervisor

Syntax:

int MAC_RemoveTrajectory(COMMUNICATION_HANDLE hCommunication, LPCTSTR cszName);

Input Parameters:

COMMUNICATION_HANDLE hCommunication	Multi-axes supervisor communication handle.
LPCTSTR cszName	Trajectory name, if * - remove all.

Return Value:

0 OK,

> 0 communication error codes:

1 The API library is not initialized;

2 Zero Communication Handle;

3 Bad Communication Handle;

4 Unable to create interface ITNElmoPublic;

5 Remove Trajectory failed;

Use MAC_GetLastError to any function that has a return value != 0 (see example) to retrieve error string.

Sample:

```
int retVal = MAC_RemoveTrajectory( hCom, _T("MyTrj") );
```

MAC_DownloadProgram

This function enables the user to download program to Multi-axes supervisor

Syntax:

```
int MAC_DownloadProgram( COMMUNICATION_HANDLE hCommunication,
                         LPCTSTR cszLocalPath );
```

Input Parameters:

COMMUNICATION_HANDLE hCommunication	Multi-axes supervisor communication handle.
LPCTSTR cszName	Trajectory name, if * - remove all.

Return Value:

int - Error value:

- 0 OK,
- > 0 communication error codes:
  - 1 The API library is not initialized;
  - 2 Zero Communication Handle;
  - 3 Bad Communication Handle;
  - 4 Unable to create interface ITNElmoPublic;
  - 5 Download program failed;

Use MAC_GetLastError to any function that has a return value != 0 (see example) to retrieve error string.

Sample:

```
int retVal = MAC_DownloadProgram( hCom, szPath )
```

MAC_RemoveProgram

This function enables the user to remove a program from the Multi-axes supervisor.

Syntax:

`int MAC_RemoveProgram`(COMMUNICATION_HANDLE hCommunication,
LPCTSTR cszName);

Input Parameters:

COMMUNICATION_HANDLE hCommunication
Multi-axes supervisor communication handle.

LPCTSTR cszName Program name, if * - remove all.

Return Value:

int - Error value:

0 OK,

> 0 communication error codes:

1 The API library is not initialized;

2 Zero Communication Handle;

3 Bad Communication Handle;

4 Unable to create interface ITNElmoPublic;

5 Remove Program failed ;

Use MAC_GetLastError to any function that has a return value != 0 (see example) to retrieve error string.

Sample:

```
int retVal = MAC_RemoveProgram( hCom, _T("*") );
```

MAC_DownloadResources

This function enables the user to define new resources for Multi-axes supervisor.

Syntax:

int MAC_DownloadResources(COMMUNICATION_HANDLE hCommunication,
LPCTSTR cszLocalPath,
BOOL bRestart);

Input Parameters:

COMMUNICATION_HANDLE hCommunication	Multi-axes supervisor communication handle.
LPCTSTR cszLocalPath	Path to resources file on desktop computer.
BOOL bRestart	Restart controller with new resources.

Return Value:

int - Error value:

0 OK,

> 0 communication error codes:

1 The API library is not initialized;
2 Zero Communication Handle;
3 Bad Communication Handle;
4 Unable to create interface ITNElmoPublic;
5 Download Resources failed;

Use MAC_GetLastError to any function that has a return value != 0 (see example) to retrieve error string.

Sample:

```
int retVal = MAC_DownloadResources( hCom, szPath, FALSE );
```

MAC_UploadLog

This function enables the user to upload log file from Multi-axes controller

Syntax:

`int MAC_UploadLog`(COMMUNICATION_HANDLE hCommunication, LPCTSTR cszLocalPath);

Input Parameters:

COMMUNICATION_HANDLE hCommunication	Multi-axes supervisor communication handle.
LPCTSTR cszLocalPath	Path to log file on desktop computer.

Return Value:

int - Error value:

- 0 OK,
- > 0 communication error codes:
  - 1 The API library is not initialized;
  - 2 Zero Communication Handle;
  - 3 Bad Communication Handle;
  - 4 Unable to create interface ITNElmoPublic;
  - 5 Upload Log failed;

Use MAC_GetLastError to any function that has a return value != 0 (see example) to retrieve error string.

Sample:

```
int retVal = MAC_UploadLog( hCom, szPath );
```

MAC_GetLastError

This function enables the user to get last error

Syntax:

`int MAC_GetLastError`(LPTSTR szError, WORD wErrorSize);

Input Parameters:

WORD wErrorSize Size of string for error receiving.

Output Parameter:

LPTSTR szError Last error message.

Return Value:

int - Error value:

>= 0	OK,
< 0	the string for reception of the error is too small, value is the necessary size of a string (with a minus).

Sample:

```
int retVal = MAC_GetLastError( szErrorCom, 1024 );
```

MAC_InitEvents

This function enables the user to initialize MAS Event mechanism

Syntax:

```
int MAC_InitEvents();
```

Return Value:

int - initialization status:

0 OK;

1 The ATL Module initialization failed;

2 Unable to create event listener thread;

Use MAC_GetLastError to any function that has a return value != 0 (see example) to retrieve error string.

Sample:

```
int nOk = MAC_InitEvents();
```

MAC_DeinitEvents

This function enables the user to close MAS Event mechanism

Syntax:

```
Void MAC_DeinitEvents();
```

Sample:

```
MAC_DeinitEvents();
```

MAC_RegCloseCallback

This function enables the user to advise/unadvise on-close-connection case user's callback

Syntax:

`int MAC_RegCloseCallback`(COMMUNICATION_HANDLE hCommunication,
MAC_ONCLOSE_CALLBACK ptrCB);

Input Parameters:

COMMUNICATION_HANDLE hCommunication
Multi-axes supervisor communication handle.

MAC_ONCLOSE_CALLBACK ptrCB – on-close-connection case user's callback.

Return Value:

int - Error value:

0 OK,

> 0 communication error codes:

1 The API library is not initialized;

2 Zero Communication Handle;

3 Bad Communication Handle;

4 Unable to create interface ITNElmoPublic;

5 TNElmo server: Dispatch event failed

Use MAC_GetLastError to any function that has a return value != 0 (see example) to retrieve error string.

Sample:

```
int nOk = MAC_RegCloseCallback( hComm, MyCloseCallback );
```

Chapter 10: The Recorder

Elmo's Recorder is used to record actions and events on the axes connected to *SimplIQ* drives. This is particularly useful with the *Maestro,* as the movements of several axes can be recorded simultaneously. The results can then be used to debug applications and to track performance.

10.1 Accessing the Recorder

The recorder can be started by clicking the recorder.exe icon. This causes the window below to open. The next step is to open a line of communication with a Maestro by clicking the connect button ( ) on the menu.

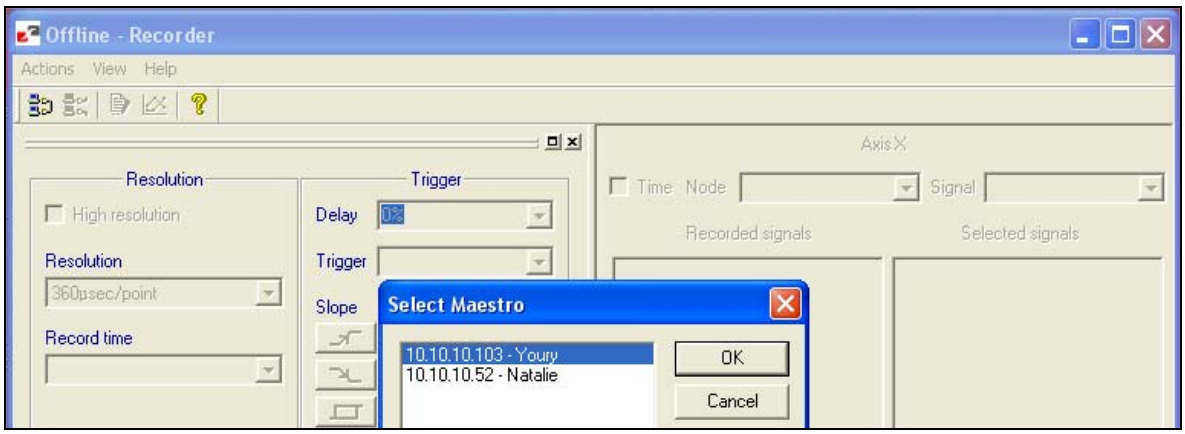


Figure 10-1 Opening a Line of Communications with a Maestro

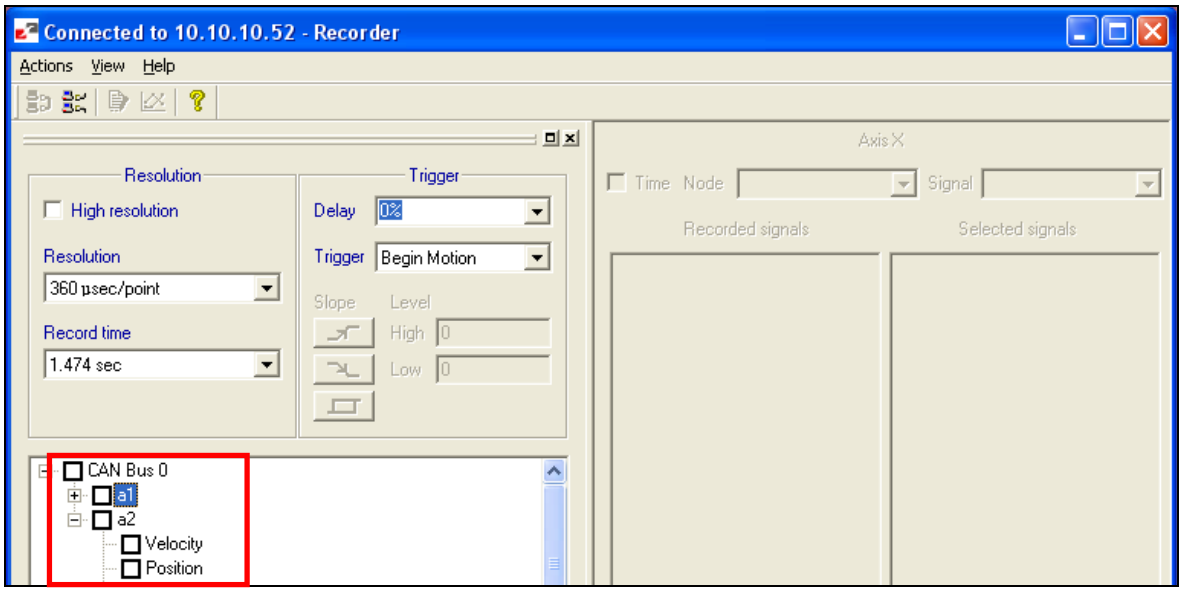
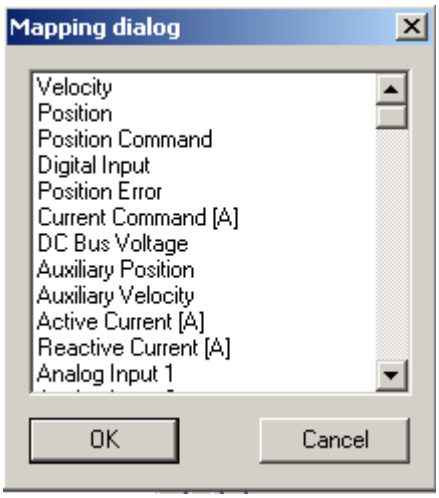


Figure 10-2 Selecting the Axis and Signals to Record



Once communications have been established, choose the Signals to record from the pick list that opens under each Axis (as shown in the figure below) or for all axes (with using the Context menu). The next step is to choose the length of time to record and the resolution of the recording. These numbers are related as there is a limit to how much can be recorded. User can to change Signals for recording by Mapping dialog (see Fig. 8-3).

The recording is "triggered" by an event. In most cases that is a Begin Motion command. Select a trigger from the **Trigger** pick list as shown below.

Figure 10-3 Signals Mapping for the Recording.

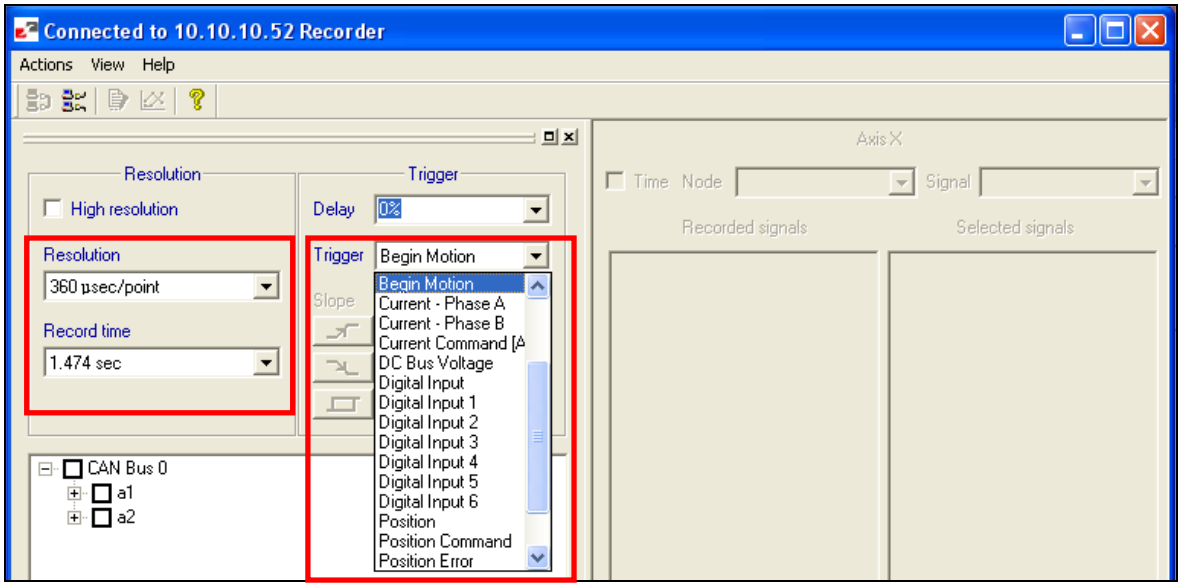


Figure 10-4 Selecting a Trigger to Start the Recording.

From the **Resolution** drop-down list, select the recording resolution, which is defined by the sampling time of the controller. You may wish to change this value depending on your current work mode: current or velocity.

From **Record Time** drop-down list, select the maximum recording interval; this is dependent upon the **Resolution** value.

In the **Level** text boxes, enter the **High** and **Low** trigger levels.

Click the appropriate button to indicate the trigger **Type**:

a) Positive slope : Set the trigger and select **High** level. The trigger will be recorded when the source signal crosses the chosen level from low to high;

b) Negative slope : Set the trigger and select **Low** level. The trigger will be recorded when the source signal crosses the chosen level from high to low;

C) Window : Set the trigger and enable **High** and **Low** levels. The trigger will be recorded when the source signal crosses the chosen levels, as follows:

- The signal crosses the Low level twice.
- The signal crosses the High level twice.
- The signal crosses the Low level once and then crosses the High level.
- The signal crosses the High level once and then crosses the Low level.

Start the recording by clicking the Record ( ) button. Recording Process Indication dialog shows the progress of the Recording procedure and wait for the recorded data resaving.

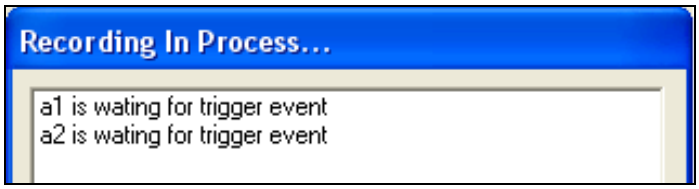


Figure 10-5 Recording Process Indication.

Go to the Elmo Studio (Fig 8-6) or Command Line Interface and start an operation:

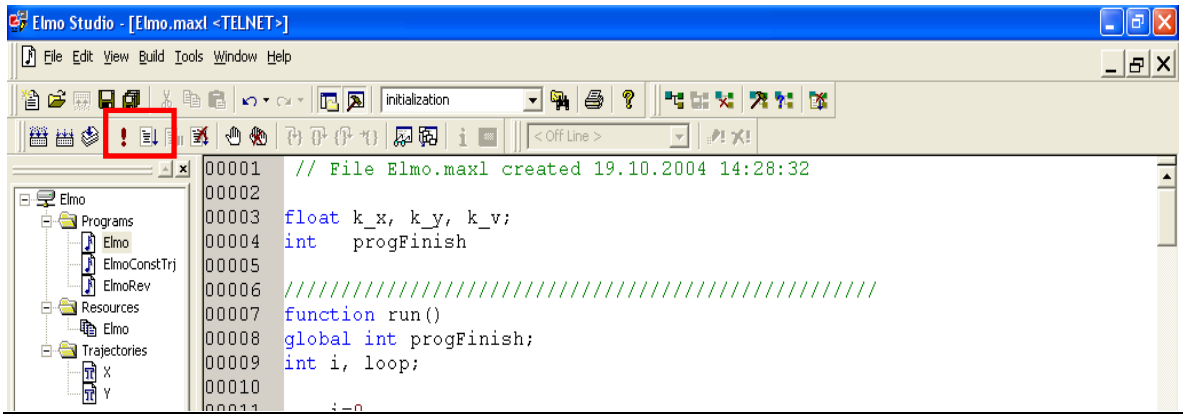


Figure 10-6 Running an Application

When the operation is complete, go to the Data Visualization Part of Recorder and select Signals combination for plotting. Some the chosen Signals can be shown in common as the uniform **Display**. As a Display Axis X time or any accessible signal can be used.

Using Add and Remove buttons (Add selected | Add display | Remove | Remove All) or the contextual menu it is possible to edit the created configuration of signals.

The current **Environment** (Device Address, set of selected signals, trigger combination and visualization settings) can be kept in a file by button Save Environment ( ) and, subsequently is loaded from a file by button Load Environment ( ).

Click the Plot Record button to display a plot of the recording by **Scope**.

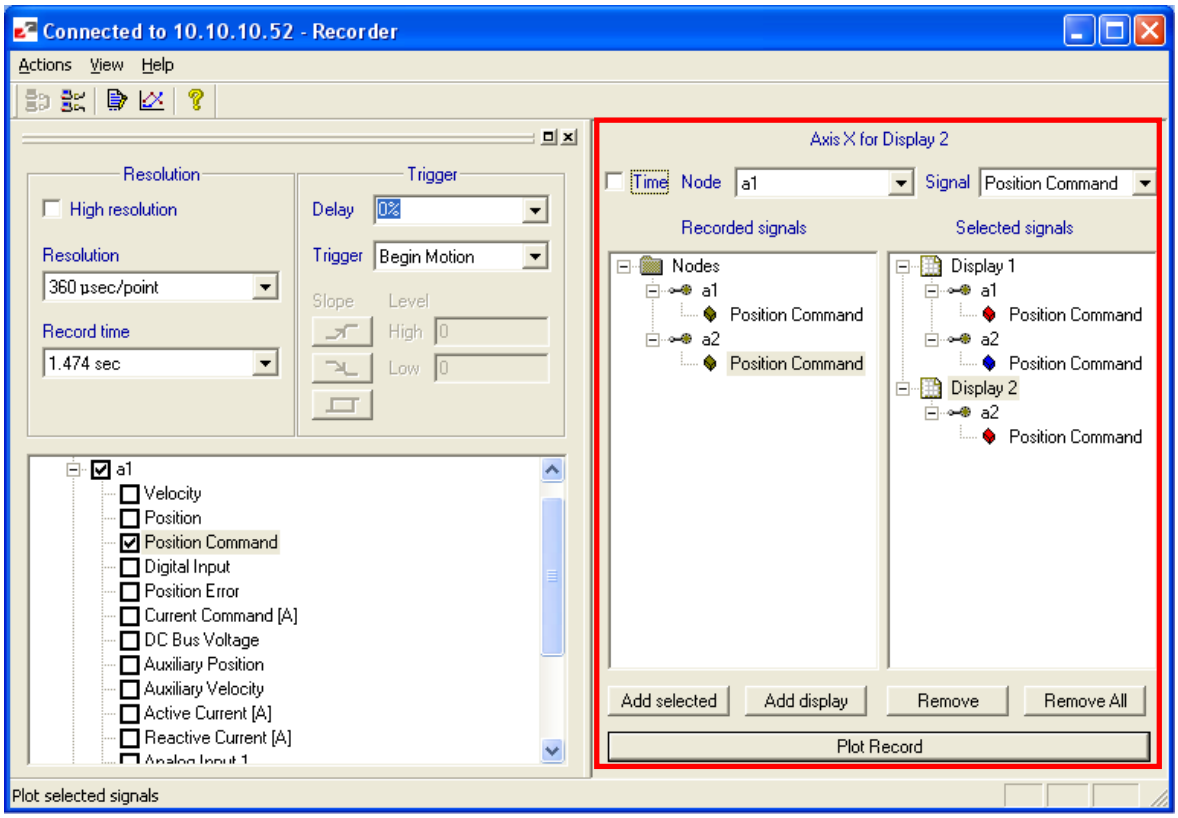


Figure 10-7 Choosing the Type of Display

The Scope is a graphic display tool that enables you to view the data that has been recorded by the motion monitor. It displays multiple recorded vectors in the same window or in separate windows, and can generate new data vectors by applying arithmetic operations on the existing data vectors. You may use the scope to view and analyze recorded motions, zooming in and out of the graphs. You may add required text to the graphs, retrieve statistical information (such as average and maximum values) and calculate step response parameters (such as bandwidth and damping).

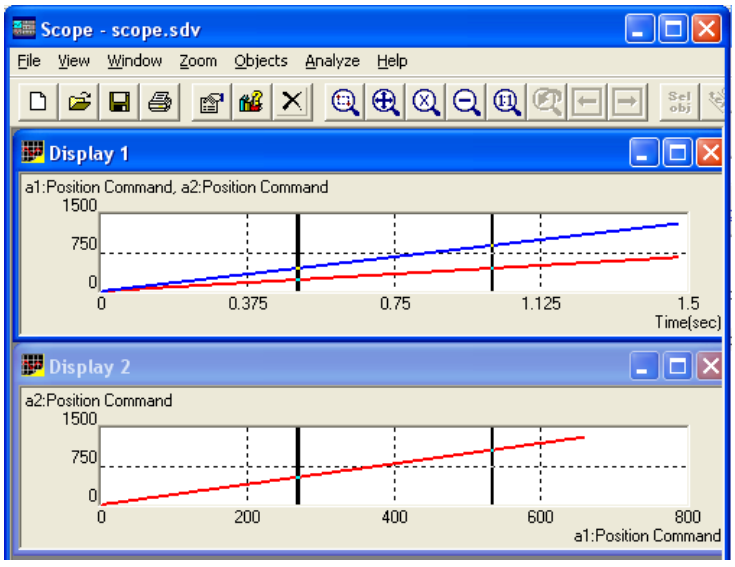


Figure 10-8 Viewing the Recorded Motion

Appendix A: Setting up the Demo Case

The *SimplIQ* Demo & Training Case is an ideal tool for developing dual-axis applications. It contains two Bassoon digital servo drives, each connected to a servo motor. The Bassoons can be programmed directly with a standard RS-232 connection. The *Maestro* can be connected via the case's CAN port.

Figure A-1 The *SimplIQ* Demo & Training Case

A.1 Setting Up the CAN nodes

For CANopen networks to function properly, every node on the network must be unique and the network must be terminated. A simple terminator is supplied with the Demo Case. If its not already in use, plug it into the last servo drive's open CANopen port.

Figure A-2 CANopen Com. Line Terminator

To ensure that the nodes are unique, connect to each Bassoon separately an check the CAN node with the ***PP[13]*** command (using the Composer's Smart_Terminal). If the nodes need resetting, use the save command (**SV**) to store them in memory.

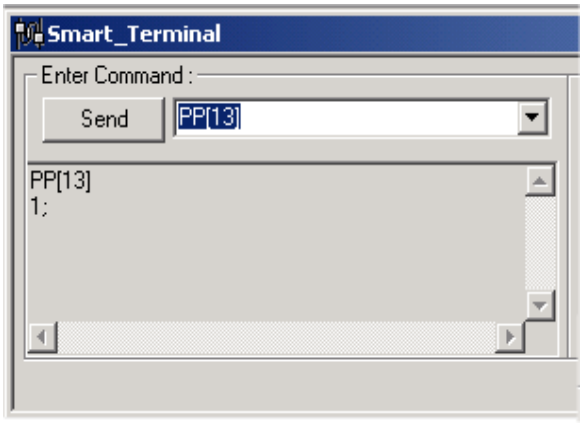


Figure A-3 Node ID Inquiry

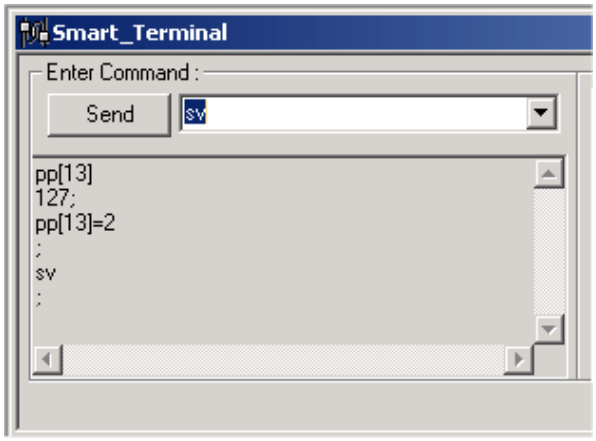


Figure A-4 Setting the Node ID

A.2 Checking the CANopen Network

To check if the CANopen network is working use the ***businfo*** command in the Maestro's CLI or Program Editor.

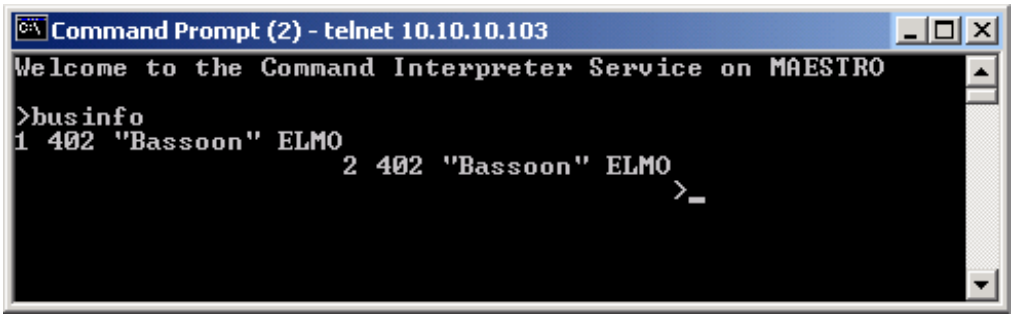


Figure A-5 Node ID Inquiry

Restart the *Maestro* if you made any changes to the Bassoons or *Maestro* software.

Appendix B: Sample Programs

B.1 Elmo Logo Outline Sample

```
// File Elmo.maxl created 19.10.2004 14:28:32
function run()
        initialization(15000,32000)
        draw()
        wait_motion()
        v1.st;
        v1.detach();
end function

function draw()
float k_x, k_y
        k_x = 1.9;
        k_y = 1.9;
        v1.starts();
        v1.vsp = 50000;//max speed
        v1.vse = 50000;//end speed

        // letter 'E'
        v1.addline  ( 0*k_x, 32000*k_y )
        v1.addcircle( 6000*k_y,90,180 )
        v1.vsp = 50000;
        v1.vse = 0;
        v1.addline  ( 15000*k_x, 20000*k_y );
        v1.addline  ( 0*k_x, 20000*k_y);
        v1.addcircle( 10000*k_y,90,180 );
        v1.vsp = 50000;
        v1.vse = 0;
        v1.addline  ( 20000*k_x, 0*k_y );

        // letter 'l'
        v1.addline  ( 30000*k_x, 20000*k_y);
        v1.addline  ( 30000*k_x, 0*k_y);
        v1.addline  ( 40000*k_x, 0*k_y);
        v1.vsp = 50000;
        v1.vse = 0;
        v1.addline  ( 50000*k_x, 20000*k_y);

        // letter 'm'
        v1.vsp = 50000;
        v1.vse = 0;
        v1.addline  ( 50000*k_x, 0*k_y);
        v1.addline  ( 50000*k_x, 15000*k_y);
        v1.addcircle( 5000*k_x, 180, -180);
        v1.vsp = 50000;
        v1.vse = 0;
        v1.addline  ( 60000*k_x, 0*k_y);
        v1.addline  ( 60000*k_x, 15000*k_y);
        v1.addcircle( 5000*k_x, 180, -180 );
        v1.addline  ( 70000*k_x, 0*k_y);

        //letter 'o'
        v1.addline  ( 80000*k_x, 15000*k_y);
        v1.addcircle( 5000*k_x, 180, -180 );
        v1.addline  ( 90000*k_x, 5000*k_y);
        v1.addcircle( 5000*k_x, 0, -180 );
        v1.vsp = 50000;
        v1.vse = 0;
        v1.addline  ( 80000*k_x, 15000*k_y);
        v1.ends();
        v1.bg;
end function
function initialization(int x_from_lim, int y_from_lim)

 int SmoothFactor
        SmoothFactor = 0;
// check UM
        if(a1.um != 5)
          a1.mo=0
          a1.um=5
        end if
        if(a2.um != 5)
          a2.mo=0
          a2.um=5
        end if

// check SF
        if(a1.sf != SmoothFactor)
          a1.mo = 0
          a1.sf = SmoothFactor
          a1.sd = 1000000000
        end if
        if(a2.sf != SmoothFactor)
          a2.mo = 0
          a2.sf = SmoothFactor
          a2.sd = 1000000000
        end if

        if(a1.mo == 0);    a1.mo=1; end if
        if(a2.mo == 0);    a2.mo=1; end if

        a1.pa = x_from_lim
        a2.pa = y_from_lim

        a1.bg; a2.bg

        while(a1.ms || a2.ms)
          wait(50)
        end while

        sync( 0, 10 );
        v1.attach();

        v1.vum = 1; // max velocity mode

        v1.vac  = 28000000

        v1.vsc  = 1; //non smoothed
        v1.vqt  = 4;

end function

//wait for motion status of vector = 0
function wait_motion()
        while( a1.ms||a2.ms )
                wait( 1 );
        end
end function
```

The Multi-Axis Language (MAXL) file examples listed below can be opened in Elmo Studio's Program Editor. These files can be modified, off-line, in File View. They can also be compiled off-line.

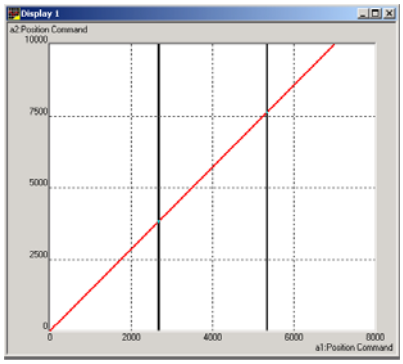
However, they cannot be downloaded and run until they communicate with a Maestro that has the associated resources. In the case of all the examples, the required resources are:

- a1: X-axis
- a2: Y-axis
- v1: vector made from a1 and a2

In other words, the program will only run if the Maestro connected by Telnet is online and has pre-defined, and on-line, a1, a2 and v2.

B.2 Line Sample

```
function run()

        sync(0, 20)
        v1.mo=0                 //motor off
        v1.px=0                 //current location is 0,0
        v1.mo=1                 // motor on
        v1.vsp=10000            //max vel
        v1.line(7000, 10000)//go to( 7K,10K)
        v1.bg                   //action!

end function
```

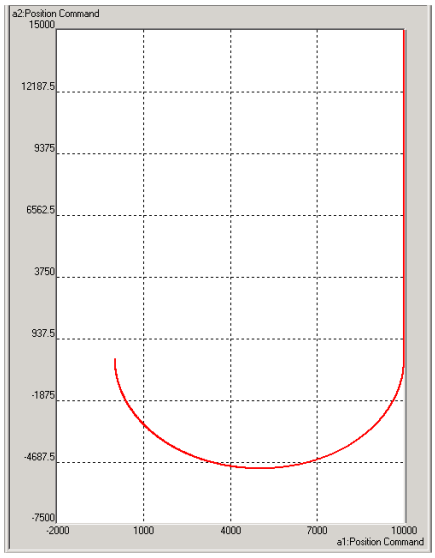


B.3 Circle Sample

```
function run()

        sync(0, 10)     //send a sync signal to CAN bus 0 every 10ms
        v1.mo=0         //motor off
        v1.px=0         //call this "home"
        v1.mo=1         //motor on
        v1.vsc=1        //smooth intersection with max. velocity

        v1.vsp=10000  // max velocity
        v1.circle(5000, 180, 90)
        v1.bg           // start motion
end function
```

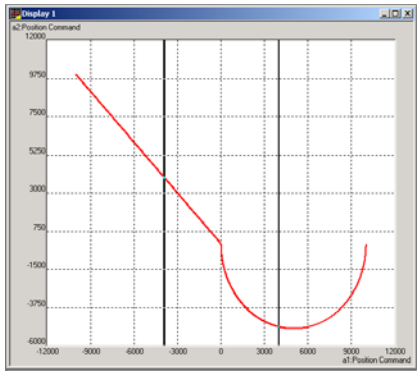
B.4 Circle-to-Line Sample

```
function run()
        sync(0, 10)         //send a sync signal to CAN bus 0 every 10ms
        v1.mo=0             //motor off
        v1.px=0             //call this "home"
        v1.mo=1             //motor on
        v1.vsc=1            //smooth intersection with max. velocity

        v1.starts()         //begin trj
        v1.vsp = 30000      //max. velocity
        v1.vse = 30000      //end velocity
        v1.addcircle( 5000,180,180 )
                    //create 5000 count radius circle from -180 to 180
        v1.vsp = 10000      //max velocity
        v1.vse = 0          //end velocity
        v1.addline( 10000,15000)
                    //create line from current point to coordinate
                    //10,000 on X and 15,000 on Y
        v1.ends()           //end trj
        v1.bg
end function
```

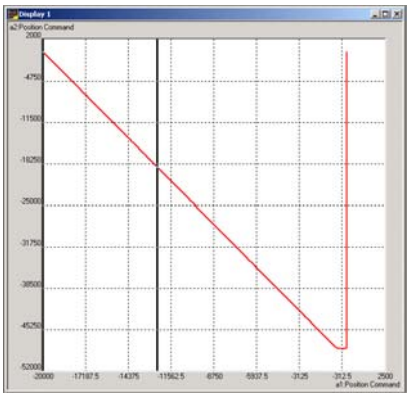


B.5 Line-to-Circle Sample

```
function run()
        sync(0, 20)         //send sync to CANbus 0 every 20ms
        v1.mo=0             // motor off
        a1.px=-10000        //x now at -10000
        a2.px=10000         //y now at 10000
        v1.mo=1             //motor on
        v1.vsc=1            //use max velocity to smooth corners

        v1.starts()         // begin building trajectory
        v1.vsp = 30000      // maximum velocity of line
        v1.vse = 30000      // end velocity of line
        v1.addline ( 0,0)   // go to (0,0)
        v1.vsp = 10000      // maximum velocity of circle
        v1.vse = 0          // end velocity of circle
        v1.addcircle( 5000,180,180 )
                            // create circle with a radius of 5000
                            // starting at 180 degrees for 180 degrees
                            // in the clockwise direction
        v1.ends()           // end trajectory build

        v1.bg             // start motion
end function
```

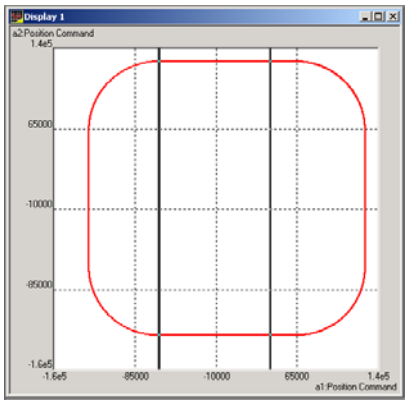


B.6 Line-to-Line Sample



```
function run()
        sync(0, 10)         //send a sync signal to CAN bus 0 every 10ms
        v1.mo=0             //motor off
        v1.px=0             //call this "home"
        v1.mo=1             //motor on
        v1.vsc = 3          //smoothed with fixed dist from the corner
        v1.vsd = 2000
                    //distance of tangent of arc from line intersection

        v1.starts()         // begin trj
        v1.vsp = 100000 //max. velocity
        v1.vse = 100000 //end velocity
        v1.addline(0, -50000)
                            //create line from current point to coordinate
                            //      0 on X and 50,000 on Y
        v1.vsp = 50000   //max. velocity
        v1.vse = 0          //end velocity
        v1.addline(-20000, 0)
                            //create line from current point to coordinate
                            //      0 on X and 50,000 on Y
        v1.ends()           // end trj

        v1.bg               //start motion
end function
```

B.7 Polygon Sample




```
function run()
        sync(0, 10)         //synchronize CAN bus 0 every 10 ms
        v1.mo=0             //motor off
        a1.px=-64000        //set coor. x to 64000
        a2.px=128000        //set coor. y to 128000
        v1.mo=1             //motor on
        v1.vsc=1            //maximum velocity smooth factor

        v1.starts()         // begin trajectory

        v1.vsp = 50000   //max velocity
        v1.vse = 50000   //end velocity

        //first line and corner
        v1.addline  ( 64000, 128000 )
        v1.addcircle( 64000,90,-90 )

        //second line and corner
        v1.addline  ( 128000, -64000 )
        v1.addcircle( 64000,0,-90 )

        //third line and corner
        v1.addline  ( -64000, -128000 )
        v1.addcircle( 64000,-90,-90 )

        //last line and corner
        v1.addline  ( -128000, 64000 )
        v1.addcircle( 64000,180,-90 )

        v1.vse = 0
        v1.ends()           //finish building trajectory
        wait(10)
        v1.bg               //start motion
end function
```

B.8 If Sample

```
function run()
        int a, b
        a=7
        b=10
        if (a < b+5)
            a=a+2
        end if
        b=b-2
end function
```

B.9 If-Else Sample

```
function run()
int a, b
        a=7
        b=10
        if (a < b+5)
            a=a+2
        else
            b=b-2
        end if
end function
```

B.10 If-Else-If Sample

```
function run()
        int a, b , c
        b=4
        c=11

        if ((b == 4)&&(c == 18))
        a=1                     // b is 4; c is 18
        else if ((b == 4)&&(c == 11))
           a=2                  // b is 4; c is 11
        else
            a=7                 // any other cases
        end if
end function
```

B.11 For Sample

```
function run()
        int i, a
        a=0

        for i=10:-2:-10         //iterate i from 10 to -10 in -2 increments
            a=a+i*2
        end for
end function
```

B.12 While Sample

```
function run()
int attribute
        while( attribute < 56)
            attribute=attribute+17
            TRACE("Attribute : %", attribute)
        end while
end function
```

B.13 Switch Sample

```
function run()
int attribute, i
        switch(attribute)
        case (1)
            i=i+i
        case (2)
            i=i*i
        case (3)
            i=i/10
        otherwise
        TRACE ("Error attribute ")
        end switch
end function
```

B.14 Break-Continue-Return Sample

```
//The "continue" statement forces immediate transfer of control to the
//     next iteration of the loop.

//The "break" statement is used to exit the loop.

//The "return" statement ends the current function
//     (and transfers control back to the calling function)

function run()
        int i,j

        for i=1:2:20
            if (i%2 == 0)
                j=i/2
                continue   //go to next iteration
            else if (i%4 == 0)
                j=i/4
                break      //exit the loop
            else
                j=i
                return     //exit this function
            end if
            i=i+1
        end for
        j=j+i
end function
```

B.15 Call Sample

```
function run()
        int multipleVar
            multipleVar=multiple(10,15) //call embedded function "multiple"
end function//after returning from the called function (multiple),
            //multipleVar=150
function [int c] = multiple(int a, int b)
            // this func has two input parameters (a and b)
            // and one output parameter (c)
        c=a*b
end function
```

B.16 Homing Sample

```
function run()
        int home_pos
        home_pos = 1000

        a1.mo = 1
            //standard Maestro homing steps are:
        a1.hm[2] = home_pos
        a1.hm[3] = 0
        a1.hm[4] = 0
        a1.hm[5] = 0
        a1.hm[1] = 1
end function
```

B.17 Program Call Sample

```
// File ProgCall.maxl
// Homing with a Call
function run()
        homing.x_pos = 0
        homing.y_pos = 0
        homing.run()

//create a line
        sync(0,10)
        v1.mo=1
        v1.vsp=10000
        v1.line(7000, 10000)
        v1.bg
end function

// File Homing.maxl
// Homing Program

int x_pos, y_pos
function run()

global int x_pos, y_pos
        // standard Maestro homing procedure
        a1.hm[2] = x_pos  //receive position from calling function
        a2.hm[2] = y_pos  //receive position from calling function

        a1.hm[3] = 0
        a2.hm[3] = 0
        a1.hm[4] = 0
        a2.hm[4] = 0

        a1.hm[5] = 0
        a2.hm[5] = 0
        a1.hm[1] = 1
        a2.hm[1] = 1
end function
```

B.18 Label and GoTo Sample

```
function run()
        int i, j

        for i = 0 : 2 : 20
        for j = 2 : 16
            if( i > j+10)
        goto ##label
            end if
        end for
        end for

##label
   i=0
end function
```

B.19 Order of Processing Sample

```
function run()
        int a , b , c, res1, res2, res3
        a = 2
        b = 4
        c = 9
        res1 = a + b * c
        res2 = a + (b * c)
        res3 = (a + b) * c
end function

        //The output from the following code is:
        //      res1 = 38
        //      res2 = 38
        //      res3 = 54
```

B.20 Global Variable Sample

```
// File GlobalVar.maxl
// Global Variables Declaration...
int a,b

// Entry point of the application.
function run()

                //Global variable declaration inside function body
                //specified by global keyword.
        global int a, b
        int c       // local variable declaration in function run()

        c=56
        b=16
        a=c+b

end function

// File GlobalVarUse.maxl created 30.01.2005 14:51:09
int pos_x, pos_y     // Global Variables Declaration...

function run()

        global int pos_x, pos_y
                //Global Var. Declaration inside a Function...
        pos_x = 1000
        pos_y = 2000
        v1.vsp = 3000
        initialization()
        v1.line(5000, 10000)
        v1.bg
end function

function initialization()
                        // Global Variables Declaration inside the Function...
                        //global int pos_x, pos_y
        sync(0,20)      //synchronize CAN devices
        v1.mo=0         //motor off
        a1.px=pos_x     //start point of Axis 1 is 1000
        a2.px=pos_y     //start point of Axis 2 is 2000
        v1.mo=1         //motor on
        v1.vum=1        //max velocity mode
end function
```

Appendix C: Performance Considerations

The Maestro can manage up to 126 axes on a standard application. However, fewer axes can be managed on applications requiring fast updates.

The Bus Load graphs show how many drives can be managed as a function of update time and bus speed.

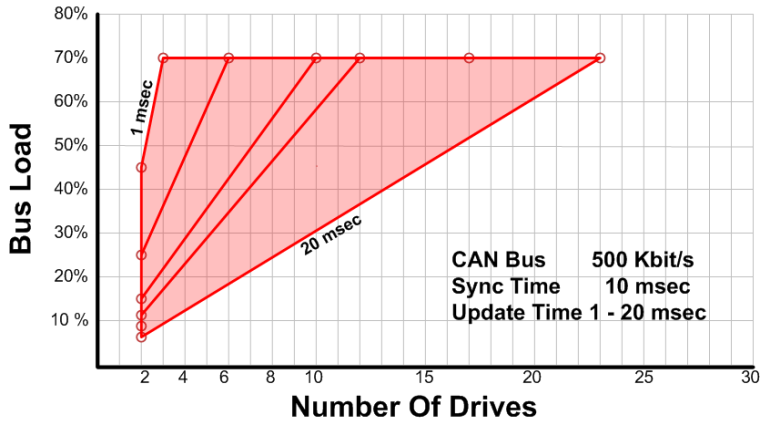


Figure C-1 Bus Load at 500Kbit/sec

For example, the Maestro can manage 4 axes on an application that requires an update every 1 msec running at a 70% load on an 800 Kbit/s bus.

On the other hand, the Maestro can manage up to 50 axes on an application that requires an update every 20 msec running at a 70% load on an 1000 Kbit/s bus.

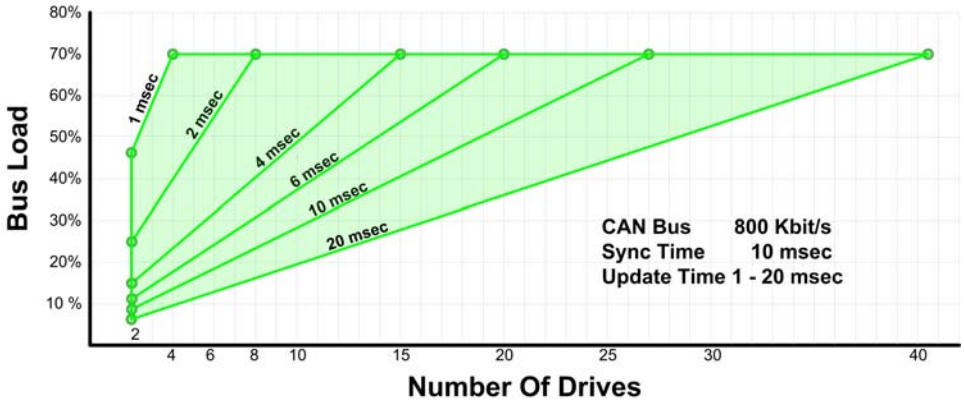


Figure C-2 Bus Load at 800Kbit/sec

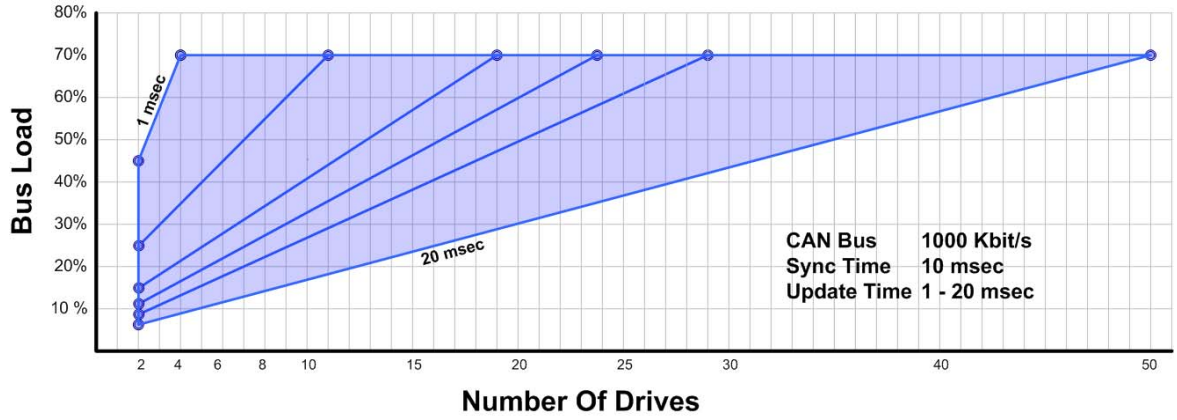


Figure C-3 Bus Load at 1000Kbit/sec

The graph above can be used as a yardstick to determine the number of updates needed to produce an acceptable path. In this case, the number of updates is a function of the diameter, speed and maximum allowable interpolation error.

For example, if the required velocity is 100,000 counts/sec along a 1000 count diameter path, and the allowable interpolation error is 10 counts, then the Maestro must perform an update every 4 msec..